1994

Application of numerical interval analysis for statistical computing in a massively parallel computing environment

Ouhong Wang
Iowa State University

Order Number 9518453

Application of numerical interval analysis for statistical computing in a massively parallel computing environment

Wang, Ouhong, Ph.D.

Iowa State University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

Application of numerical interval analysis for statistical computing in a massively parallel computing environment

by

Ouhong Wang

A Dissertation Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Department: Statistics
Major: Statistics

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1994

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

For almost all computational problems in statistics, obtaining self-validating results is not an easy task. What we obtain after performing computation in the traditional way are just approximations to the theoretically true values. Without special computational arrangements we do not know how close they are to the true values. Sometimes they are accurate enough for our applications. Other times they are too inaccurate to be satisfactory. But there is absolutely no indication of which is the case for a certain problem. Two major computational errors contribute to this uncertainty. The first one is rounding errors. All the computers we use employ finite number systems, which means many real numbers are not representable. So we are forced to do rounding and truncating all the time. The accumulated errors in some cases may be serious enough to put the computed results far away from where they should be. The other kind of computational error is approximation error. This occurs when we have no direct method to carry out our computation, and we use approximation in the design of the algorithm. An example is the use of a Taylor expansion to a finite number of terms in computing e^x. Hence it is the algorithm itself that will not give us accurate results. For traditional computation, these two kinds of errors are usually both present, and we will always ask the question "how accurate is my result?"

Interval analysis is a tool to deal with the above mentioned two sources of errors. It is a relatively new and growing branch of applied mathematics. It is an approach to computing that treats an interval as a computing element. We replace a scalar with an interval that surely contains the scalar real number. This way we take care of all uncertainty of initial parameter values. In designing the algorithm we use interval analysis to count for both the approximation and the error. Throughout the computation we design special treatments for roundings to insure that the theoretically true scalar value is carried over from one interval-valued computing step to the next. The final result is also expressed in the form of an interval with the guarantee that the true value is contained in it. Hence, computations in properly implemented interval arithmetic, together with analysis of algorithms from interval analysis point of view, produce approximations to the true values as well as guaranteed error bounds. Thus, we have at the outset, a general mechanism for bounding the accumulation of both roundoff errors and approximation errors.

The use of interval analysis in statistics has two major applications. The first one is to get highly accurate results in some computationally intensive problems, e.g., multivariate probability computation. These are usually very difficult problems and require extremely good computer software support to obtain satisfactory results. There are software packages which attempt to deal with this kind of problem, but they usually do not converge for high dimensional integrations, and when they do converge, the results are not reliable. The use of interval analysis will guarantee any pre-specified accuracy for a wide class of problems. Another application for interval analysis is global optimization, which is widely used in statistics, e.g., optimal design, nonlinear least squares, etc. None of the currently existing software systems can

guarantee convergence to a global optimum. They may very well converge to just a local optimum. But there is no reliable mechanism in the algorithms employed by these systems to tell whether a solution is a global or local optimum. However, using interval analysis, local optima are easily distinguished from global optima and the latter are normally located with relative ease within a finite starting region. Interval analysis takes into account the whole starting region and is guaranteed to find any global optima within that region.

Since interval analysis is very time consuming because more computation is involved, we use a parallel machine of SIMD architecture throughout this research to help get good performance. Experience shows that we can get satisfactory results within reasonable running time.

In this dissertation we review the basic elements of interval analysis and propose interval methods on the parallel machine for the above mentioned statistical applications. Chapter 2 introduces the elements of interval analysis. Chapter 3 describes the architecture of the parallel machine and how to do interval analysis on it. The usage of the machine is illustrated with an application to survey sampling analysis. In chapter 4 we present parallel interval methods for self-validating computation of multivariate Normal and multivariate t probabilities. Chapter 5 proposes parallel interval optimization methods, illustrated by examples from optimal design, least squares and time series. And finally chapter 6 contains some conclusions.

CHAPTER 2. ELEMENTS OF INTERVAL ANALYSIS

Interval analysis was first introduced by Moore (1962). A useful description of early work in this area is given in his monograph (Moore, 1979). The basic idea is to extend the usual real arithmetic and employ closed bounded intervals of real numbers as the basic numeric elements.

We will presently introduce arithmetic operations for interval numbers and develop methods for computing more complicated functions and for integrations, on which our statistical applications will be based.

From now on, real numbers will be denoted by lower case letters, intervals by capital letters and endpoints of an interval by the base letter accompanied by an under/over-score. Thus $X = [\underline{X}, \overline{X}]$ denotes the interval X having real number endpoints $\underline{X}$ and $\overline{X}$.

2.1 Interval Arithmetic and Properties

Suppose we have intervals X and Y. Then for scalars x and y such that

$$\underline{X} \leq x \leq \overline{X} \quad \text{and} \quad \underline{Y} \leq y \leq \overline{Y}, \tag{2.1}$$

we have $\underline{X} + \underline{Y} \leq x + y \leq \overline{X} + \overline{Y}$. Thus, if we define $X + Y$ to be all the possible summations of x and y satisfying 2.1, then $X + Y$ is another interval with lower

endpoint $\underline{X} + \underline{Y}$ and upper endpoint $\overline{X} + \overline{Y}$. Similarly, we can define subtraction, multiplication, reciprocal and division, and they are summarized as follows:

$$\begin{aligned}
X + Y &= \{x + y | x \in X, y \in Y\} \\
&= [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}] \\
X - Y &= \{x - y | x \in X, y \in Y\} \\
&= [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}] \\
XY &= \{xy | x \in X, y \in Y\} \\
&= [\min(\underline{X}\underline{Y}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{X}\overline{Y}), \max(\underline{X}\underline{Y}, \underline{X}\overline{Y}, \overline{X}\underline{Y}, \overline{X}\overline{Y})] \\
1/Y &= \{1/y | y \in Y\}, \quad 0 \notin Y \\
&= [1/\overline{Y}, 1/\underline{Y}] \\
X/Y &= \{x/y | x \in X, y \in Y\}, \quad 0 \notin Y \\
&= X(1/Y)
\end{aligned}$$

The following algebraic properties of interval arithmetic are immediate consequences of the set theoretic definitions of the interval arithmetic:

$$\begin{aligned}
X + (Y + Z) &= (X + Y) + Z \\
X(YZ) &= (XY)Z \\
X + Y &= Y + X \\
XY &= YX
\end{aligned}$$

for any intervals X,Y,Z. Specially, if we use a single real number to represent a degenerate interval, an interval with identical endpoints, we have

$$0 + X = X + 0 = X$$

$$0X = X0 = 0$$

$$1X = X1 = X$$

for any X.

Thus, addition and multiplication are *associative* and *commutative.* However, the distributive law does not always hold. For example, we have

$$[1,2] \cdot (1-1) = 0;$$

whereas

$$[1,2] \cdot 1 - [1,2] \cdot 1 = [-1,1] \neq 0.$$

Thus, $X(Y+Z) = XY + XZ$ is not always true.

We do, however, always have the following algebraic property

$$X(Y+Z) \subseteq XY + XZ. \tag{2.2}$$

We call this property *subdistributivity.* As can be seen, it is really a combination of algebraic *and* set theoretic relations.

In certain special cases, distributivity holds. Some particularly useful cases are

$$\begin{aligned} &x(Y+Z) = xY + xZ \quad \text{for } x \text{ real; } Y, Z \text{ intervals,} \\ &X(Y+Z) = XY + XZ \quad \text{if } YZ > 0. \end{aligned} \tag{2.3}$$

Thus, we can distribute multiplication by a real number and we can distribute multiplication by any interval through sums of intervals all of the same sign. The properties 2.2 and 2.3 follow easily from the definitions of interval arithmetic.

Note that $X - X = 0$ and $X/X = 1$ only when X is a degenerate interval, because actually $X - X = [\underline{X} - \overline{X}, \overline{X} - \underline{X}]$, and $X/X = [\underline{X}/\overline{X}, \overline{X}/\underline{X}]$ for $0 < \underline{X}$, and $X/X = [\overline{X}/\underline{X}, \underline{X}/\overline{X}]$ for $\overline{X} < 0$. The cancellation law holds for interval addition:

$$X + Z = Y + Z \quad \text{implies} \quad X = Y.$$

2.2 Rounded Interval Arithmetic

Interval arithmetic cannot be done by a digital computer because the computer cannot represent all real numbers. Computers employ floating-point values which all represent rational numbers in a particular finite range. We can, however, make a useful substitution for interval arithmetic by employing what is called rounded interval arithmetic.

Given a real interval $X = [\underline{X}, \overline{X}]$, an expression of this interval in the computer can be made as $\hat{X} = [fl(\underline{X}), fl(\overline{X})]$, where $fl(\underline{X})$ is the biggest floating-point number such that $fl(\underline{X}) \leq \underline{X}$. Similarly, $fl(\overline{X})$ is the smallest floating-point number such that $fl(\overline{X}) \geq \overline{X}$. We have $X \subseteq \hat{X}$ provided that it is indeed possible to find the required floating-point values. If $\underline{X}$ or $\overline{X}$ is outside the range of real numbers spanned by the set of floating-point numbers, then it is impossible to express $\hat{X}$.

Rounded interval arithmetic is carried out using intervals whose endpoints are floating-point numbers. Directed roundings are employed, rounding toward $-\infty$ when computing lower endpoint, and rounding toward $+\infty$ where computing upper interval endpoint. The objective is to obtain, at each stage of the interval computations, an interval having floating-point endpoints which contains the true real interval which would be obtained if it were possible to do real arithmetic. This objective will be reached provided that no underflow or overflow occurs when the floating-point arithmetic operations are completed.

A numerical example will illustrate the distinction between interval arithmetic and rounded interval arithmetic.

Let

$$\begin{aligned} X &= [-0.613E10^{-2}, -0.610E10^{-2}] \\ Y &= [+0.100E10^{+1}, +0.300E10^{+1}] \\ Z &= X(1 + 1/Y). \end{aligned}$$

We compute Z first in exact interval arithmetic. We have

$$\begin{aligned} Z &= X(1 + 1/[1,3]) \\ &= X(1 + [1/3, 1]) \\ &= X[4/3, 2] \\ &= [-0.1226E10^{-1}, -0.8133\cdots E10^{-2}]. \end{aligned}$$

Next, we compute Z using rounded interval arithmetic based on signed three decimal digit mantissas and floating point number representation. We have

$$1 = [+0.100E10^{+1}, +0.100E10^{+1}],$$

$$1/Y \subseteq [+0.333E10^{+0}, +0.100E10^{+1}],$$

$$1 + 1/Y \subseteq [+0.133E10^{+1}, +0.200E10^{+1}],$$

$$X[+0.133E10^{+1}, +0.200E10^{+1}] \subseteq [-0.123E10^{-1}, -0.811E10^{-2}].$$

Each of the four numerical intervals on the right hand sides of the above relations is the smallest (narrowest) interval representable in the chosen number form containing the quantity on the left. The final result contains the exact interval value of Z. The containing interval is slightly wider than Z because of the cumulative effect of dropping digits beyond the third one after the decimal point. Hence we are making the computed interval slightly wider, but all the real numbers resulted from

the definition of interval arithmetic are contained in it. By carrying enough digits we can come as close as we please to the exact interval arithmetic results.

2.3 Interval Inclusions

Extension of the concept of a real-valued function of one or more real variables to the case of an interval-valued function of one or more interval variables is not trivial. Here we will describe the techniques used in interval analysis to link these two conceptually different areas.

The basic purpose for using interval analysis is to simultaneously obtain function information over a region. So what we are interested in, for any given scalar function f, is what is called the united extension.

Definition 2.1: The *United Extension* of f is

$$\overline{f}(X_1, \cdots, X_n) = \bigcup_{(x_1, \cdots, x_n) \in (X_1, \cdots, X_n)} \{f(x_1, \cdots, x_n)\}.$$

In some cases this may not be easy to obtain. For example, the function f may not be continuous, hence $\overline{f}(X_1, \cdots, X_n)$ can not be expressed in the form of an interval. So in general, we will think in terms of an interval inclusion of f.

Definition 2.2: An *Interval Inclusion* of f over the region $(X_1, X_2, \cdots, X_n)$ is an interval valued function F such that

$$\overline{f}(X_1, \cdots, X_n) \subseteq F(X_1, \cdots, X_n).$$

We will be interested in how to compute interval inclusions for given functions. We need the following definitions.

Definition 2.3: An interval-valued function F of the interval variables $X_1, \cdots, X_n$ is *Inclusion Monotonic* if

$$Y_i \subseteq X_i \qquad \text{implies} \qquad F(Y_1, \cdots, Y_n) \subseteq F(X_1, \cdots, X_n).$$

Definition 2.4: An *Interval Extension* of f is an interval-valued function F of n interval variables X_i with the property

$$F(x_1, ..., x_n) = f(x_1, ..., x_n)$$

for real arguments.

We have the following theorem due to Moore (1979) which provides a way to find an interval inclusion for a scalar function f.

Theorem 2.1: If F is an inclusion monotonic interval extension of f, then F is an interval inclusion of f.

2.3.1 Finite Evaluation

The evaluation of a rational function requires only a finite number of arithmetic steps. This fact provides a method to obtain an interval inclusion of any given rational function. To see this first note that *united extensions*, which all have the subset property, are inclusion monotonic. Hence, since the interval arithmetic functions are united extensions of the real arithmetic functions $(+, -, \times, /)$, we have the result that interval arithmetic as defined in Section 2.1 is inclusion monotonic:

$$Y_1 \subseteq X_1 \qquad \text{and} \qquad Y_2 \subseteq X_2$$

implies

$$Y_1 + Y_2 \subseteq X_1 + X_2,$$
$$Y_1 - Y_2 \subseteq X_1 - X_2,$$
$$Y_1 Y_2 \subseteq X_1 X_2,$$
$$Y_1 / Y_2 \subseteq X_1 / X_2.$$

From the transitivity of the partial order relation, $\subseteq$, it follows, by finite induction, that rational interval functions are inclusion monotonic. Now, given any rational function f, if we replace all the variables and operations with the corresponding interval elements to obtain F, obviously it is an interval extension of f by definition. We call an F thus obtained a *Natural Interval Extension* of f. Since F is rational, we just showed it is inclusion monotonic. Then by Theorem 2.1, F is an interval inclusion of f. Thus in general, for any rational function, if we replace all computing elements with the corresponding interval elements, the resulting interval-valued function will be an interval inclusion of the original scalar function. It is actually an easy and routine procedure.

We have to notice, however, that this procedure does not imply uniqueness as we see from the following example. Let

$$f(x) = 1 - 5x + \frac{x^3}{3}.$$

Consider the natural interval extension

$$P(X) = 1 - 5X + \frac{1}{3} X \cdot X \cdot X$$

and another interval extension

$$Q(X) = 1 - X(5 - \frac{1}{3} X \cdot X).$$

It is readily verified that $P([2,3]) = [-34/3, 0]$ and $Q([2,3]) = [-10, -3]$. Each interval does, of course, contain the range of $f(x)$ over $[2,3]$, but one is a tighter inclusion than the other. This example illustrates what is generally true for polynomial functions, namely the Horner form $A_0 + X(A_1 + X(A_2 + \cdots + XA_n)\cdots)$ is never wider than the natural interval extension sum of powers $A_0 + A_1X + A_2XX + \cdots + A_nXX\cdots X$. This is due to subdistributivity of interval arithmetic. The example also illustrates that usually we have to pay attention to the choice of inclusion in order to obtain the best available interval inclusion.

2.3.2 Irrational Evaluation

Irrational functions require an infinite number of steps of arithmetic, and the above method does not apply. We must deal with irrational functions individually and take into consideration the special properties of each. Since we can obtain interval inclusions for rational functions, our general approach will be to approximate the irrational function by a rational function, and attempt to obtain inclusions for both the approximation and the remainder.

Some methods were suggested in early years of interval analysis for monotone functions. For example, for e^X, it was suggested that we could compute an interval inclusion of it as $\left[e^{\underline{X}}, e^{\overline{X}}\right]$. Although we could not compute the required endpoint values exactly, we could extend the machine-computed endpoints outward by an amount just great enough to include the approximation error. The problem with it is that the method is highly hardware and software dependent. We have to analyze how hardware performs arithmetic and how e^x is approximated in order to extend outward enough to accommodate all the approximation errors. Since every machine

is different and every software for e^x is different, this method is incompatible and it requires more general methods for such functions.

The irrational functions that will be used later in this dissertation are exponential and logarithm functions. We will provide methods to obtain interval inclusions for them as a demonstration of the general method.

Suppose we have an interval X and want to get an interval inclusion of e^X. We can approximate e^x by a rational function, and one way of doing this is to use the Taylor expansion. We will consider only positive x's because $e^{-x} = 1/e^x$. For a scalar $x > 0$, we have

$$\begin{aligned} e^x &= 1 + x + \frac{x^2}{2!} + \cdots + \frac{x^n}{n!} + R_n \\ &= M_n + R_n \end{aligned}$$

where M_n is called the rule part and R_n is the remainder.

M_n is rational, so we can easily get an interval inclusion of it using the method in Section 2.3.1. For the remainder, notice that

$$\begin{aligned} R_n &= \frac{x^{n+1}}{(n+1)!} + \frac{x^{n+2}}{(n+2)!} + \frac{x^{n+3}}{(n+3)!} + \cdots \\ &= a_{n+1} + a_{n+2} + a_{n+3} + \cdots, \end{aligned}$$

and we can find an n such that

$$\begin{aligned} \frac{a_{n+2}}{a_{n+1}} &= \frac{x}{n+2} < \frac{x}{n+1} < 1 \\ \frac{a_{n+3}}{a_{n+2}} &= \frac{x}{n+3} < \frac{x}{n+1} < 1 \end{aligned}$$

$$\vdots$$

Hence,

$$\begin{aligned} R_n &= a_{n+1} + a_{n+2} + a_{n+3} + \cdots \\ &< a_{n+1}\left[1 + \frac{x}{n+1} + \left(\frac{x}{n+1}\right)^2 + \cdots\right] \\ &= \frac{a_{n+1}}{1 - \frac{x}{n+1}}. \end{aligned}$$

Since $x > 0$, we have

$$0 < R_n < \frac{\frac{x^{n+1}}{(n+1)!}}{1 - \frac{x}{n+1}}.$$

Note that the upper bound for R_n is rational. If we use interval analysis to compute the right hand side and denote its upper endpoint by placing a $\triangle$ in front of it, then

$$0 < R_n < \triangle\frac{\frac{X^{n+1}}{(n+1)!}}{1 - \frac{X}{n+1}},$$

which yields an interval inclusion for R_n. Together with interval inclusion for M_n, and the fact that operation "+" is inclusion monotonic, we have

$$e^X \subseteq 1 + X + \frac{X^2}{2!} + \cdots + \frac{X^n}{n!} + \left[0, \triangle\frac{\frac{X^{n+1}}{(n+1)!}}{1 - \frac{X}{n+1}}\right],$$

which is an interval inclusion for e^X.

The logarithm function can be handled similarly. Consider $ln(Y)$ for an interval Y. We will assume $y \geq 1$ because $ln(y) = -ln(1/y)$ for $y < 1$.

For scalar $y \geq 1$, transform according to $x = (y-1)/(y+1)$, then

$$ln(y) = ln\left(\frac{1+x}{1-x}\right) \quad \text{and} \quad 0 \leq x < 1.$$

The Taylor expansion of $ln(y)$ in terms of x is

$$\begin{aligned} ln(y) &= 2\left(x + \frac{x^3}{3} + \frac{x^5}{5} + \cdots + \frac{x^{2n+1}}{2n+1} + R_n\right) \\ &= 2(M_n + R_n), \end{aligned}$$

where

$$R_n = \frac{x^{2n+3}}{2n+3} + \frac{x^{2n+5}}{2n+5} + \cdots = a_{2n+3} + a_{2n+5} + \cdots.$$

Note that M_n is rational. The remainder R_n can be bounded as follows:

$$\frac{a_{2n+5}}{a_{2n+3}} = \frac{2n+3}{2n+5}x^2 < x^2 < 1$$
$$\frac{a_{2n+7}}{a_{2n+5}} = \frac{2n+5}{2n+7}x^2 < x^2 < 1$$
$$\vdots$$

$$\Longrightarrow R_n < a_{2n+3}(1 + x^2 + x^4 + \cdots)$$
$$= \frac{a_{2n+3}}{1-x^2}$$
$$= \frac{\frac{x^{2n+3}}{2n+3}}{1-x^2}.$$

Similar as in the case of e^X,

$$ln(Y) \subseteq 2\left(X + \frac{X^3}{3} + \frac{X^5}{5!} + \cdots + \frac{X^{2n+1}}{2n+1} + \left[0, \triangle\frac{\frac{X^{2n+3}}{2n+3}}{1-X^2}\right]\right)$$

is an interval inclusion for $ln(Y)$.

It is worth mentioning that the above derived interval inclusions are not unique. In fact, we can rather easily think of a method which yields a better inclusion. For example, we can take advantage of the fact that $ln(y)$ is monotone. To obtain an interval inclusion for $ln(Y)$, define two degenerate intervals $[\underline{Y}, \underline{Y}]$ and $[\overline{Y}, \overline{Y}]$. Applying the above formula twice will give interval enclosures of $ln(\underline{Y})$ and $ln(\overline{Y})$. These intervals will have very small width when n is large. The interval inclusion of $ln(Y)$ is then obtained by taking the obvious lower and upper endpoints from the two intervals. This will result in a very tight enclosure of $ln(Y)$.

2.4 Interval Integration

Integration plays an important role in multivariate probability computation. Because in our later applications we put a lot of effort on expanding integrand functions in Taylor expansions, here we will derive methods for integration of interval valued continuous functions which are in Taylor expansion forms.

For a continuous interval valued function F, there are two continuous real valued functions $\underline{F}$ and $\overline{F}$ such that, for real t, we have $F(t) = [\underline{F}(t), \overline{F}(t)]$. In this case, interval integration is defined as

$$\int_{[a,b]} F(t)dt = \left[\int_{[a,b]} \underline{F}(t)dt \ , \ \int_{[a,b]} \overline{F}(t)dt\right]. \tag{2.4}$$

See Moore (1979).

From this definition we can easily see that interval integration preserves inclusion, which means: if $F(t) \subseteq G(t)$ for all $t \in [a,b]$, then

$$\int_{[a,b]} F(t)dt \subseteq \int_{[a,b]} G(t)dt. \tag{2.5}$$

Now, consider a univariate function $f(x)$. What we need is an interval inclusion for $\int_{[a,b]} f(x)dx$. Suppose $X = [a,b]$ and c is the center point of $[a,b]$. Then

$$\begin{aligned} f(x) &= f(c) + f'(c)(x-c) + \cdots + f^{(n-1)}(c)\frac{(x-c)^{n-1}}{(n-1)!} + f^{(n)}(\xi)\frac{(x-c)^n}{n!}, \xi \in X \\ &\in f(c) + f'(c)(x-c) + \cdots + f^{(n-1)}(c)\frac{(x-c)^{n-1}}{(n-1)!} + F^{(n)}(X)\frac{(x-c)^n}{n!}. \end{aligned}$$

where $F^{(n)}$ is an interval inclusion of the nth order derivative $f^{(n)}$. By 2.5,

$$\begin{aligned} \int_a^b f(x)dx \in & \int_a^b \left(f(c) + f'(c)(x-c) + \cdots + f^{(n-1)}(c)\frac{(x-c)^{n-1}}{(n-1)!} + F^{(n)}(X)\frac{(x-c)^n}{n!} \right) dx \\ = & \sum_{i=0}^{n-1} f^{(i)}(c) \left. \frac{(x-c)^{i+1}}{(i+1)!} \right|_a^b + \int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx \end{aligned}$$

$$\subseteq 2\sum_{\substack{i=0\\ even}}^{n-1} F^{(i)}(C)\frac{H^{i+1}}{(i+1)!} + \int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx, \tag{2.6}$$

where $C = [c,c]$ and $H = [h,h]$ where h is the half length of X. Note that $F^{(n)}(X)$ is actually a constant interval. Let $F^{(n)}(X) = [s,t]$, then

- if n is even, for any $x \in [a,b]$,

$$F^{(n)}(X)\frac{(x-c)^n}{n!} = \left[s\frac{(x-c)^n}{n!}, t\frac{(x-c)^n}{n!}\right], \text{ since } (x-c)^n \text{ is positive.}$$

By 2.4, we have

$$\begin{aligned}
\int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx &= \left[\int_a^b s\frac{(x-c)^n}{n!}dx, \int_a^b t\frac{(x-c)^n}{n!}dx\right] \\
&= \left[s\left.\frac{(x-c)^{n+1}}{(n+1)!}\right|_a^b, t\left.\frac{(x-c)^{n+1}}{(n+1)!}\right|_a^b\right] \\
&= \left[2s\frac{h^{n+1}}{(n+1)!}, 2t\frac{h^{n+1}}{(n+1)!}\right] \\
&= 2[s,t]\frac{h^{n+1}}{(n+1)!} \\
&= 2F^{(n)}(X)\frac{h^{n+1}}{(n+1)!} \\
&\subseteq 2F^{(n)}(X)\frac{H^{n+1}}{(n+1)!}.
\end{aligned}$$

- if n is odd,

$$\int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx = \int_a^c [s,t]\frac{(x-c)^n}{n!}dx + \int_c^b [s,t]\frac{(x-c)^n}{n!}dx.$$

For the first part,

$$[s,t]\frac{(x-c)^n}{n!} = \left[t\frac{(x-c)^n}{n!}, s\frac{(x-c)^n}{n!}\right], \text{ since } (x-c)^n \text{ is negative.}$$

Hence,

$$\begin{aligned}\int_a^c [s,t]\frac{(x-c)^n}{n!}dx &= \left[\int_a^c t\frac{(x-c)^n}{n!}, \int_a^c s\frac{(x-c)^n}{n!}\right] \\ &= \left[t\,\frac{(x-c)^{n+1}}{(n+1)!}\Big|_a^c, s\,\frac{(x-c)^{n+1}}{(n+1)!}\Big|_a^c\right] \\ &= \left[-t\frac{h^{n+1}}{(n+1)!}, -s\frac{h^{n+1}}{(n+1)!}\right].\end{aligned}$$

For the second part,

$$[s,t]\frac{(x-c)^n}{n!} = \left[s\frac{(x-c)^n}{n!}, t\frac{(x-c)^n}{n!}\right], \text{ since } (x-c)^n \text{ is positive.}$$

Hence,

$$\begin{aligned}\int_c^b [s,t]\frac{(x-c)^n}{n!}dx &= \left[\int_c^b s\frac{(x-c)^n}{n!}, \int_c^b t\frac{(x-c)^n}{n!}\right] \\ &= \left[s\,\frac{(x-c)^{n+1}}{(n+1)!}\Big|_c^b, t\,\frac{(x-c)^{n+1}}{(n+1)!}\Big|_c^b\right] \\ &= \left[s\frac{h^{n+1}}{(n+1)!}, t\frac{h^{n+1}}{(n+1)!}\right].\end{aligned}$$

So for odd n,

$$\begin{aligned}\int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx &= \left[-t\frac{h^{n+1}}{(n+1)!}, -s\frac{h^{n+1}}{(n+1)!}\right] + \left[s\frac{f^{n+1}}{(n+1)!}, t\frac{h^{n+1}}{(n+1)!}\right] \\ &= \left[(s-t)\frac{h^{n+1}}{(n+1)!}, (t-s)\frac{h^{n+1}}{(n+1)!}\right] \\ &= [s,t]\frac{h^{n+1}}{(n+1)!} - [s,t]\frac{h^{n+1}}{(n+1)!} \\ &= F^{(n)}(X)\frac{h^{n+1}}{(n+1)!} - F^{(n)}(X)\frac{h^{n+1}}{(n+1)!} \\ &\subseteq F^{(n)}(X)\frac{H^{n+1}}{(n+1)!} - F^{(n)}(X)\frac{H^{n+1}}{(n+1)!}.\end{aligned}$$

Formally, we can combine the even and odd cases for n, write it as

$$\int_a^b F^{(n)}(X)\frac{(x-c)^n}{n!}dx \subseteq F^{(n)}(X)\frac{H^{n+1}}{(n+1)!} - F^{(n)}(X)\frac{(-H)^{n+1}}{(n+1)!}.$$

Hence from 2.6 we get an interval inclusion for the whole integration:

$$\int_a^b f(x)dx \in 2\sum_{\substack{i=0 \\ even}}^{n-1} F^{(i)}(C)\frac{H^{i+1}}{(i+1)!} + F^{(n)}(X)\frac{H^{n+1}}{(n+1)!} - F^{(n)}(X)\frac{(-H)^{n+1}}{(n+1)!}. \quad (2.7)$$

In later applications we will derive interval inclusions for multivariate integrations in a similar way.

CHAPTER 3. PARALLEL COMPUTING ENVIRONMENT

We can see from the previous chapter that interval analysis requires much more computation than traditional arithmetic. For rational functions it may be as much as four times the number of floating point operations. For irrational functions it is much more than that because we can not use the built-in optimized routines, instead we have to write our own subroutine for each one of them. So we are actually trading running time for accuracy. In this research parallel computers were used to achieve reasonable running time. We will introduce, in this chapter, the architecture of the machines we used and how interval analysis can be implemented in this particular environment. And finally, we will present an example of how to take advantage of the parallel machines.

3.1 MasPar Architecture

The computers we used are MasPar machines. The architecture is Single Instruction Multiple Data (SIMD). We have access to two MasPar machines: MP1 and MP2. MP1 has four times the number of processors than MP2, and the architectures are very similar. We will focus on MP1 here for simplicity.

MP1 consists of 2 major subsystems: Front End and Data Parallel Unit (DPU). Front End is a Unix Subsystem, actually a DEC 5000/240 workstation. It deals with

usual tasks like user interface, file management, and so on. DPU is the heart of MP1. It has two parts: ACU and PE Array. ACU is the Array Control Unit. It controls the PE Array, determines what data and instructions should be sent to the PE Array, and performs non-parallel part of the computation. The PE Array is the Processing Element Array. Each PE has its own processor with 40 32-bit registers and 16K bytes of RAM. MP1 has 16384 processors, and they are arranged as a 128×128 grid with toroidal wrap. At any given time, a PE can be either active or inactive. All the active PEs have to execute the same instruction at that time, but they process different data. This is basically what SIMD means.

On MP1, data is identified as either singular or plural. A singular variable x resides on ACU and does not have anything to do with the PE Array. A plural variable y resides on each PE node and may have different values on different PE nodes.

An important aspect of a parallel machine is the communication among processors. On MP1 this is done by two special constructs: *XNet* and *Router*.

The *XNet* construct allows each active PE to communicate with a PE that is a uniform distance and direction from the active PE. The direction must be one of the following: *N E S W NE SE SW NW*. For example,

$$px = xnetS[3].py;$$

means that all active PEs simultaneously fetch the value of their south neighbor's *py* 3 distance away and store it into their *px*.

The *Router* construct allows each active PE to communicate with any other PE in an arbitrary way. For example,

$$k = router[i].j;$$

means that the value of j is fetched from the ith PE and then stored into k for all the active PEs. Since i is plural, it can have different value on each PE, the PEs may be fetching different values of j. Apparently *Router* is more flexible than *XNet*, but it is about 8 times slower. Figure 3.1 shows part of the PE Array and the difference between *Xnet* and *Router*. Relative to the center PE node, all the PE nodes can be accessed by *Router* but only the shaded PE nodes can be accessed by *XNet*.

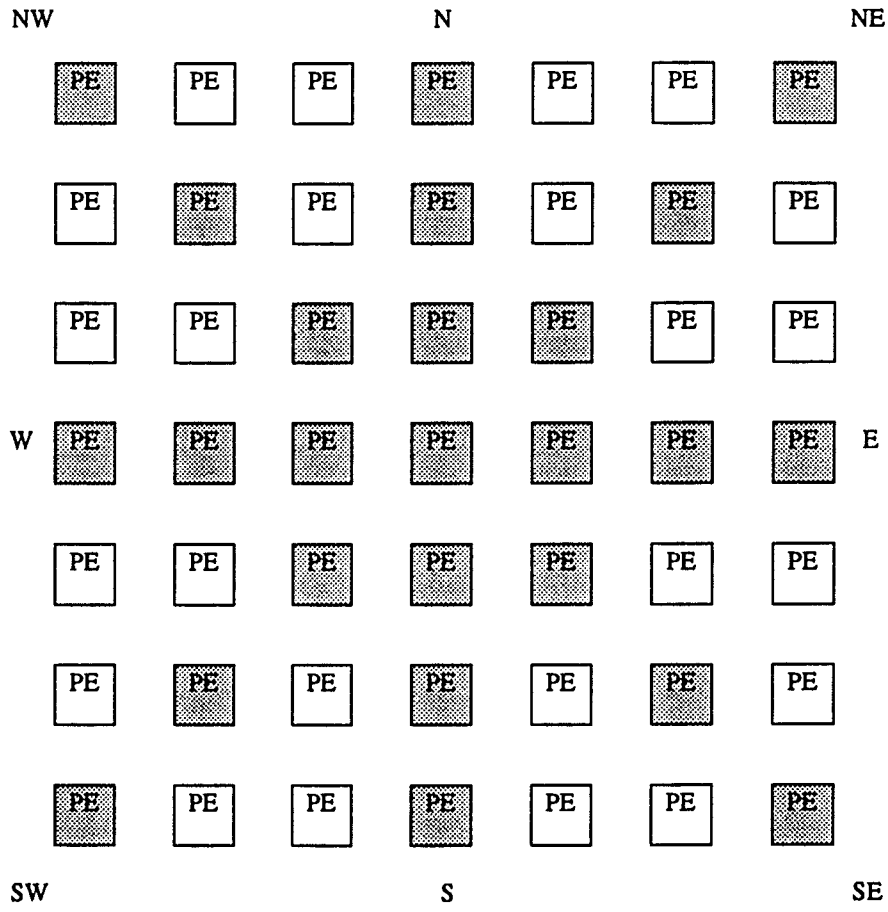

Figure 3.1: Part of the PE Array of the MasPar machine

3.2 Implementation of Interval Routines on MasPar

Both MP1 and MP2 are used in our research. But the ways they implement rounded interval arithmetic are quite different. This is because MP2 supports IEEE standard for rounding mode control with floating point arithmetic, while MP1 does not. This is the only noticeable difference between our application on these two machines. So on MP1, we have to control the roundings in software. The ideas are mainly from Clemmesen (1984). We will introduce first the MP1 routines, then the MP2 routines.

Below are detailed descriptions on implementation of various routines for MP1. Note that they are all plural routines. In other words they can run simultaneously on all PE nodes.

- *succ* : For a double precision number, it will give the next representable double precision floating point number in MP1's number system. In order to accomplish this, the interval type on MP1 is defined as

```
typedef union  {  double unf;
                  unsigned long uni[2]
               }  singleton;
typedef struct {  singleton l;
                  singleton r
               }  intvl;
```

  Since double precision variable *unf* and integer variable *uni[2]* occupy the same memory space, we can add 1 to the last bit of the integer representation of double precision type and get the next floating point number.

- *pred*: Similar to *succ*, but it will subtract 1 from the last bit to give the previous number.

- *add*: $u+v$ is implemented by making use of $u+v = u \oplus v + ((u \ominus (u \oplus v)) \oplus v)$ (Knuth (1981), p221), where "+" gives the theoretically true value, and "$\oplus$" gives the round-to-nearest value. We know that on MP1 the rounding mode is fixed at round-to-nearest. So we first compute $((u \ominus (u \oplus v)) \oplus v)$. If it is positive, then $\triangle(u+v) = succ(u \oplus v)$. If it is negative, $\triangle(u+v) = u \oplus v$. $\triangledown(u+v)$ can be obtained in a similar way.

- *sub* : Implemented via $u+v$ because $(u-v) = u+(-v)$.

- *mul* : No formula as for $u+v$ exists for $u \times v$. Hence we always perform $\triangle(u \times v) = succ(u \otimes v)$, $\triangledown(u \times v) = pred(u \otimes v)$. The result may not be the tightest inclusion, but it does include the true value.

- *div* : same idea as in *mul.*

On MP2, the above routines are simpler. We just set the rounding mode as desired and then perform the arithmetic. We do not need routines *succ* and *pred*, but we added two more function routines for square and square root.

- *squ* : Computes X^2 for interval X. It performs $X \times X$ first, then if the lower end point of it is negative, set it to zero. It is easily verified that this procedure will give the tightest inclusion.

- *sqt* : Performs $\sqrt{X}$ for interval X. Because there is no document to explain how square root is done on MP2, we can not just set the rounding mode and

then call the subroutine. Instead, we designed our own using interval Newton methods. For reference see Moore (1979). In short, for an interval inclusion of the root of a scalar function $f(x)$, if $X_{(0)}$ contains the theoretical scalar root of $f(x)$, then we have the following iterative method :

$$X_{(k+1)} = X_{(k)} \bigcap N(X_{(k)}), \quad k = 0, 1, 2, \cdots$$

where

$$N(X_{(k)}) = m(X) - f(m(X))/F'(X),$$

and $m(X)$ is the midpoint of interval X, F' is an inclusion for the derivative of f. If we apply this to function $x^2 - c$, we will obtain an interval Newton method for computing $\sqrt{c}$:

$$X_{(k+1)} = X_{(k)} \bigcap \left\{ m(X_{(k)}) - \left(m\left(X_{(k)}\right)^2 - c \right) / 2X_{(k)} \right\}.$$

To obtain an $X_{(0)}$ that contains $\sqrt{c}$, we can use the intrinsic scalar square root function. The iteration will continue until the length of $X_{(k+1)}$ is smaller than a pre-specified tolerance.

3.3 Example: Survey Sampling Analysis on MasPar

In this section we want to use an example to demonstrate how to take advantage of the MasPar machine. The example we use is survey sampling analysis. A traditional software package for survey sampling analysis is PC CARP, which uses serial algorithms. For huge data sets, which are not unusual in survey sampling, running

time becomes a very serious problem. Some of the analyses of PC CARP are implemented on MasPar, and the potential utility of the SIMD type digital computers for survey analysis is thus investigated.

In PC CARP, the subroutine to get totals and the estimated covariance matrix for specified variables forms the main algorithm. We will focus on the implementation of just this analysis.

Let $y_{ijk} = \{y_{ijk1}, y_{ijk2}, \cdots, y_{ijkp}\}$ denote the vector of variables to be analyzed, where $i = 1, 2, \cdots, L$; $j = 1, 2, \cdots, n_i$; $k = 1, 2, \cdots, m_{ij}$; i is the stratum identification, j is the cluster identification, and k is the element-within-cluster identification. y_{ijkr} is the ijk-th observation for the r-th variable.

The estimated total for variable number r is $\hat{y}_r = \sum_{i=1}^{L} \sum_{j=1}^{n_i} \sum_{k=1}^{m_{ij}} w_{ijk} y_{ijkr}$, $r = 1, 2, \cdots, p$, where w_{ijk} is the weight for the ijk-th observation. The expression is simple enough and does not need further work.

The estimated covariance matrix for $\hat{y} = \{\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_p\}$ is

$$\hat{V}\{\hat{y}\} = \sum_{i=1}^{L} (n_i - 1)^{-1} n_i (1 - f_i) \sum_{j=1}^{n_i} \left(d_{ij.} - \bar{d}_{i..}\right)' \left(d_{ij.} - \bar{d}_{i..}\right) \tag{3.1}$$

where $d_{ij.} = \{d_{ij.1}, d_{ij.2}, \cdots, d_{ij.p}\}$, $d_{ij.r} = \sum_{k=1}^{m_{ij}} w_{ijk} y_{ijkr}$, $\bar{d}_{i..} = n_i^{-1} \sum_{j=1}^{n_i} d_{ij}$ and f_i is the sampling rate for the i-th stratum.

The inner-most summation is the variance-covariance matrix for i-th stratum taking cluster totals as individual values. This is a two-pass algorithm. First pass gets the means, second pass gets the variance-covariance matrix. For large data sets, two-pass methods may be too time consuming compared with one-pass methods.

Here we consider an alternative way of computing $\sum_{j=1}^{n_i} \left(d_{ij.} - \bar{d}_{i..}\right)' \left(d_{ij.} - \bar{d}_{i..}\right)$. This is a $p \times p$ matrix. The st-th element of it is V_{n_i}, where V_{n_i} is computed using

the following one-pass accumulative method :

$$V_h = V_{h-1} + \left(h \times d_{ih.s} - \sum_{l=1}^{h} d_{il.s}\right)\left(h \times d_{ih.t} - \sum_{l=1}^{h} d_{il.t}\right)/h(h-1)\,. \qquad (3.2)$$

This result is due to Youngs and Cramer (1971). According to the authors' testing, 3.2 is faster and only slightly less accurate than the two-pass method and is the best among candidate one-pass methods.

The data is read in to MasPar in batches. For each batch, some computation is done to accumulate totals and the covariance matrix. Each batch of the data consists of 16384 observations so that each observation is distributed to a PE node. On each PE, $w_{ijk}y_{ijkr}$ is formed and the value is summed over the whole PE Array to update the total $\hat{y}_r$.

For the covariance matrix update, because the total number of variables is unlimited, the covariance matrix can be huge. Since the memory space on PE Array is very limited, the whole covariance matrix must be divided into several pieces and updated one by one. When we update a piece, from 3.2, we have to get cluster totals. With one observation on each PE, we can simultaneously determine which nodes are ends of clusters, then accumulate the cluster totals to those nodes. After this we can easily update the current piece of the covariance matrix according to 3.2. Because we are processing 16384 observations at a time, instead of just one at a time, the running time is shortened significantly. Table 3.1 is a comparison of MasPar running time and PC CARP running time. Note that the entry format is MasPar time/386-25 time in seconds. Since PC CARP can deal with at most 50 variables, the entries corresponding to number of variables = 80, 100, 120 are just MasPar running times alone.

Table 3.1: Comparison of running times for total analysis on MP1 and on PC

Number of Variables	Number of Observations				
	1000	5000	10000	20000	29189
2	0.3/4	1/19	2/37	4/73	5/106
10	1/12	3/58	5/113	12/226	15/326
20	3/36	6/177	10/347	22/697	28/1016
30	6/60	11/300	16/590	32/1173	44/1718
40	9/85	15/422	23/829	48/1655	59/2417
50	13/111	20/548	29/1082	58/2141	80/3100
80	25	37	52	108	138
100	36	50	67	141	180
120	49	66	84	181	230

CHAPTER 4. MULTIVARIATE PROBABILITY COMPUTATION

Multivariate probability computation is a very computationally intensive problem, especially when the number of dimensions of the problem is large. There are existing software systems to help solve this problem, e.g. the specially designed library named QUADPACK (Piessens *et al.*, 1983). Other software vendors, like IMSL, also provide multivariate integration routines which can be used for probability computation (IMSL, 1986). These routines use traditional numerical analysis methods. For low-dimensional problems they usually work well. But for higher than five dimensional problems they frequently fail. Even if they converge, the results are usually not very reliable. Basically for high-dimensional problems we do not have any reliable analytical methods for evaluating integrals. So people often turn to Monte Carlo methods (Davis and Robinowitz, 1975). By nature this method is not aiming at high accuracy and is very time consuming. Hence the problem we are facing is to achieve convergence, accuracy and quickness in a single algorithm. The use of interval analysis, together with the MasPar machine, will enable us to obtain highly accurate results within reasonable running time. In this chapter we will present parallel interval integration methods for multivariate Normal and multivariate t distributions over finite regions.

4.1 Multivariate Normal Distribution

Consider the multivariate Normal probability over a hyper-cube :

$$I = \int_{a_1}^{b_1} \int_{a_2}^{b_2} \cdots \int_{a_n}^{b_n} \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} exp\left(-\frac{1}{2}\left(\underline{x} - \underline{\mu}\right)' \Sigma^{-1} \left(\underline{x} - \underline{\mu}\right)\right) d\underline{x}. \qquad (4.1)$$

For this special integration, Schervish (1984) designed an algorithm for n <8, which used a locally adaptive numerical integration strategy based on Newton-Cotes 2,3 or 5 point rules. In most cases it worked well for $n < 5$. But the running time is very long. There are cases with n=5 that it will not terminate within 50 hours. Also it was designed for only low-dimensional problems.

Genz (1992) designed a general algorithm. After a set of transformations, the final form becomes :

$$I = (e_1 - d_1) \int_0^1 (e_2 - d_2) \cdots \int_0^1 (e_m - d_m) \int_0^1 d\underline{w}. \qquad (4.2)$$

where

$$\begin{aligned} d_i &= \Phi\left(\left(a_i - \sum_{j=1}^{i-1} c_{ij} \Phi\left(d_j + w_j(e_j - d_j)\right)\right) / c_{ii}\right), \text{and} \\ e_i &= \Phi\left(\left(b_i - \sum_{j=1}^{i-1} c_{ij} \Phi(d_j + w_j(e_j - d_j))\right) / c_{ii}\right). \end{aligned}$$

Here Φ is the standard univariate Normal CDF, c_{ij} is the ij-th element of C where CC^T is the Cholesky decomposition of Σ. This form was then used as input for a Monte Carlo method and a subregion adaptive method. 4.2 may look more complicated than 4.1, but the author claimed that it had some advantage over 4.1: because $e_i - d_i, i > 1$ depends on w_1; $e_i - d_i, i > 2$ depends on w_1 and w_2, etc, this actually defines a priority order for $w_1, w_2, \cdots, w_m$, and this information can be used in determining how to efficiently split the current region into small pieces. Test results

showed some improvement over Schervish's method. For $n \leq 10$, for some special cases, it can produce multivariate normal probabilities accurate to two or three decimal digits in two or three seconds on a DECstation 3100. But in general, there is no guarantee of the level of accuracy that can be achieved.

Using interval analysis and MasPar, we can solve these multivariate integration problems more quickly and with much greater accuracy.

4.1.1 Trivariate Normal Case

Let's first look at the trivariate Normal integration. To simplify things we assume the inverse of Σ is already known, and the distribution is already centered. Hence we have the following problem :

$$I = k_0 \int_{a_1}^{b_1} \int_{a_2}^{b_2} \int_{a_3}^{b_3} f(x_1, x_2, x_3) dx_3 dx_2 dx_1, \tag{4.3}$$

where

$$k_0 = \frac{1}{(2\pi)^{3/2} |\Sigma|^{1/2}}, \quad \Sigma^{-1} = \begin{bmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{bmatrix},$$

$$f(x_1, x_2, x_3) = e^{g(x_1, x_2, x_3)}, \quad \text{and} \quad g(x_1, x_2, x_3) = -\frac{1}{2} \underline{x}' \Sigma^{-1} \underline{x}.$$

Note that the integration region is finite in every dimension.

Let $\sqsupset m \sqsubset$ denote all the possible partitions of m into three parts m_1, m_2 and m_3, and let

$$X_i = [a_i, b_i], \quad c_i = \frac{a_i + b_i}{2}, \quad h_i = \frac{b_i - a_i}{2}.$$

We can expand $f(x_1, x_2, x_3)$ in a Taylor expansion at the center point $\underline{c} = (c_1, c_2, c_3)'$ of the integration region :

$$f(\underline{x}) = \sum_{k=0}^{m-1} \left[\sum_{\exists k \sqsubset} \frac{1}{k_1!k_2!k_3!} \frac{\partial^k}{\partial {x_1}^{k_1} \partial {x_2}^{k_2} \partial {x_3}^{k_3}} f(\underline{c}) (x_1 - c_1)^{k_1} (x_2 - c_2)^{k_2} (x_3 - c_3)^{k_3} \right]$$
$$+ \sum_{\exists m \sqsubset} \frac{1}{m_1!m_2!m_3!} \frac{\partial^m}{\partial {x_1}^{m_1} \partial {x_2}^{m_2} \partial {x_3}^{m_3}} f(\xi_1, \xi_2, \xi_3)(x_1 - c_1)^{m_1} (x_2 - c_2)^{m_2} (x_3 - c_3)^{m_3},$$

where $\xi_i \in X_i, i = 1, 2, 3$, and $\underline{x} = (x_1, x_2, x_3)'$. Consequently,

$$f(\underline{x}) \in \sum_{k=0}^{m-1} \left[\sum_{\exists k \sqsubset} \frac{1}{k_1!k_2!k_3!} \frac{\partial^k}{\partial {x_1}^{k_1} \partial {x_2}^{k_2} \partial {x_3}^{k_3}} f(\underline{c}) (x_1 - c_1)^{k_1} (x_2 - c_2)^{k_2} (x_3 - c_3)^{k_3} \right]$$
$$+ \sum_{\exists m \sqsubset} \frac{1}{m_1!m_2!m_3!} \frac{\partial^m}{\partial {x_1}^{m_1} \partial {x_2}^{m_2} \partial {x_3}^{m_3}} f(X_1, X_2, X_3)(x_1 - c_1)^{m_1} (x_2 - c_2)^{m_2} (x_3 - c_3)^{m_3}.$$

Then by iteratively applying the same method utilized to derive 2.7 in each of the three dimensions, we obtain

$$I \in K_0 \cdot 2^3 \sum_{\substack{k=0 \\ even}}^{m-1} \sum_{\substack{\exists k \sqsubset \\ k_i even}} \frac{1}{(k_1+1)!(k_2+1)!(k_3+1)!} \frac{\partial^k}{\partial {x_1}^{k_1} \partial {x_2}^{k_2} \partial {x_3}^{k_3}} F(\underline{C}) {H_1}^{k_1+1} {H_2}^{k_2+1} {H_3}^{k_3+1}$$
$$+ K_0 \sum_{\exists m \sqsubset} \frac{1}{(m_1+1)!(m_2+1)!(m_3+1)!} \Big(\Big(\Big(\frac{\partial^m}{\partial {x_1}^{m_1} \partial {x_2}^{m_2} \partial {x_3}^{m_3}} F(X_1, X_2, X_3) {H_1}^{m_1+1}$$
$$- \frac{\partial^m}{\partial {x_1}^{m_1} \partial {x_2}^{m_2} \partial {x_3}^{m_3}} F(X_1, X_2, X_3)(-H_1)^{m_1+1} \Big)^1 {H_2}^{m_2+1} - \binom{1\,1}{} (-H_2)^{m_2+1} \Big)^2$$
$$ {H_3}^{m_3+1} - \binom{2\,2}{} (-H_3)^{m_3+1} \Big)^3. \qquad (4.4)$$

Here K_0 is an interval inclusion of k_0, H_i is the degenerate interval $[h_i, h_i]$, partial derivatives on F are interval inclusions of partial derivatives on f, and parenthesis with numbers on top are used to avoid rewriting the content within their previous appearances.

The expression on the right side of 4.4 is an easy to deal with interval expression in the sense that it only involves rational functions, provided that we know how to compute interval inclusions of high order partial derivatives. Section 2.3.1 outlined a

straight forward method of getting interval inclusions for rational functions. Hence, if we can enclose partial derivatives, we can easily obtain an interval inclusion for I. Now the problem becomes how to compute high order partial derivatives and get interval inclusions of them.

4.1.2 Automatic Differentiation

Let's focus on the computation of interval inclusions of $\frac{\partial^k}{\partial {x_1}^{k_1} \partial {x_2}^{k_2} \partial {x_3}^{k_3}} f$ with $k = k_1 + k_2 + k_3$. The idea of automatic differentiation is to find a systematic way of computing the values of all orders of partial derivatives evaluated at the same point. The process is numeric and does not require the explicit algebraic expression of the derivatives.

For a general function f, suppose it is analytic in some neighborhood of $\underline{x}_0$. Define

$$\begin{aligned} (f)^{k_1 \cdots k_n} &= \frac{\partial^{k_1 + \cdots + k_n}}{\partial {x_1}^{k_1} \cdots \partial {x_n}^{k_n}} f(\underline{x}_0), \quad \text{and} \\ (f)_{k_1 \cdots k_n} &= \frac{1}{k_1! \cdots k_n!} (f)^{k_1 \cdots k_n}. \end{aligned}$$

We want to derive methods for computing $(f)_{k_1 k_2 \cdots k_n}$. Although $(f)^{k_1 k_2 \cdots k_n}$ and $(f)_{k_1 k_2 \cdots k_n}$ differ only by a constant factor, we choose to deal with $(f)_{k_1 k_2 \cdots k_n}$ because the derived formula will be simpler.

Corollary 4.1: For n-variate analytic functions u and v, we have the following *Generalized Leibniz's Rule*:

$$(uv)_{k_1 k_2 \cdots k_n} = \sum_{j_1=0}^{k_1} \cdots \sum_{j_n=0}^{k_n} (u)_{j_1 \cdots j_n} (v)_{k_1 - j_1 \cdots k_n - j_n}$$

Proof: Focusing on just one of the n variables and using the usual Leibniz's Rule

$$(uv)^{(k_i)} = \sum_{j=0}^{k_i} \binom{k_i}{j} (u)^{(j)} (v)^{k_i - j},$$

it's easy to prove

$$(uv)_{k_i} = \sum_{j=0}^{k_i} (u)_j (v)_{k_i - j}$$

for variable number i.

Then for the multivariate case,

$$\begin{aligned}
(uv)_{k_1 k_2 \cdots k_n} &= \left(\cdots \left(\left((uv)_{k_1 0 \cdots 0} \right)_{0 k_2 0 \cdots 0} \right) \cdots \right)_{0 \cdots 0 k_n} \\
&= \left(\cdots \left(\left(\sum_{j_1=0}^{k_1} (u)_{j_1 0 \cdots 0} (v)_{k_1 - j_1 0 \cdots 0} \right)_{0 k_2 0 \cdots 0} \right) \cdots \right)_{0 \cdots 0 k_n} \\
&= \left(\cdots \left(\sum_{j_1=0}^{k_1} \left((u)_{j_1 0 \cdots 0} (v)_{k_1 - j_1 0 \cdots 0} \right)_{0 k_2 0 \cdots 0} \right) \cdots \right)_{0 \cdots 0 k_n} \\
&= \left(\cdots \left(\sum_{j_1=0}^{k_1} \sum_{j_2=0}^{k_2} (u)_{j_1 j_2 0 \cdots 0} (v)_{k_1 - j_1 k_2 - j_2 0 \cdots 0} \right) \cdots \right)_{0 \cdots 0 k_n} \\
&= \cdots \\
&= \sum_{j_1=0}^{k_1} \cdots \sum_{j_n=0}^{k_n} (u)_{j_1 \cdots j_n} (v)_{k_1 - j_1 \cdots k_n - j_n}. \qquad \square
\end{aligned}$$

Apply these to the computation of $(f)_{k_1 k_2 k_3}$ where f is as in 4.3. The k_i's can not be all zero. Suppose $k_1 \geq 1$. We can take the derivatives in two steps : first take the first order partial derivative of f with respect to x_1; then take all the remaining partial derivatives on the resulting expression. We get

$$(f)_{100} = (f)^{100} = (e^g)^{100} = f g'_{x_1}$$

and

$$(f)_{k_1 k_2 k_3} = \frac{1}{k_1} (f_{100})_{k_1 - 1, k_2, k_3} = \frac{1}{k_1} (f g'_{x_1})_{k_1 - 1, k_2, k_3}.$$

Applying the Generalized Leibniz's Rule, we find

$$(f)_{k_1k_2k_3} = \frac{1}{k_1}\sum_{i=0}^{k_1-1}\sum_{j=0}^{k_2}\sum_{l=0}^{k_3}(f)_{ijl}(g'_{x_1})_{k_1-1-i,k_2-j,k_3-l}. \tag{4.5}$$

If $k_1 = 0$ and $k_2 \geq 1$, or $k_1 = k_2 = 0$ and $k_3 \geq 1$, similar formulas will be obtained by first taking the first order partial derivative of f with respect to x_2 or x_3 instead of x_1. So from now on we will talk about only 4.5, but the other two cases are implied.

Equation 4.5 is actually an iterative form. We are building up a partial derivative based on the lower order partial derivatives. And the very basic values to start with are

$$(f)_{000} = f, \quad (f)_{100} = fg'_{x_1}, \quad (f)_{010} = fg'_{x_2}, \quad (f)_{001} = fg'_{x_3}. \tag{4.6}$$

They are all evaluated at the same point.

To compute $\frac{\partial^k}{\partial {x_1}^{k_1}\partial {x_2}^{k_2}\partial {x_3}^{k_3}}F$ in 4.4, we only need to find an interval inclusion of 4.5. Here f is an exponential function, but we will have no problem obtaining its first few derivatives in 4.6 using the results in Section 2.3.2. g is a polynomial, and so are its partial derivatives. Hence for 4.5, which is in the form of rational computation, we can replace everything with its interval counterpart and obtain an interval inclusion for $(f)_{k_1k_2k_3}$. This is $\frac{\partial^m}{\partial {x_1}^{m_1}\partial {x_2}^{m_2}\partial {x_3}^{m_3}}F$.

Note that 4.5 applies to partial derivatives all evaluated at the same point. If we want partial derivative values at different points, 4.5 will not be suitable. But this suffices for our purpose because in 4.4 we need derivatives evaluated at only two points. Hence we can build two series of partial derivatives, one for point (C_1, C_2, C_3) and the other for point (X_1, X_2, X_3).

4.1.3 General Case

For the general n case the expression of I will be quite similar to 4.4, and it can be derived in a similar manner. Everything is still rational as in 4.4, so we can get an interval inclusion of I, provided that we can get interval inclusions of the partial derivatives of the n-variate f. The iterative form of the partial derivatives of f can be expressed as

$$(f)_{k_1\cdots k_n} = \frac{1}{k_p}\sum_{l_1=0}^{k_1}\cdots\sum_{l_p=0}^{k_p-1}\cdots\sum_{l_n=0}^{k_n}(f)_{l_1\cdots l_n}(g'_{x_p})_{k_1-l_1,\cdots,k_p-1-l_p,\cdots,k_n-l_n}, \tag{4.7}$$

where p is such that $k_1 = k_2 = \cdots = k_{p-1}$ and $k_p \neq 0$. We will say $(f)_{k_1\cdots k_n}$ is from k family if $k_1 + k_2 + \cdots + k_n = k$. The iterative computing procedure will be : first get $(f)_{k_1\cdots k_n}$ for 0 family, which is f itself, then for 1 family, then for 2 family, and so on.

The number of terms in 4.7 increases rapidly as k increases. But if we look at the special form of the function we are dealing with, things will simplify greatly. Similar to 4.3, we have

$$g(x_1,\cdots,x_n) = -\frac{1}{2}\underline{x}'\Sigma^{-1}\underline{x}$$

and

$$g'_{x_p} = -\underline{t}'_p\underline{x}$$

where $\underline{t}_p$ is the $p^{\underline{th}}$ column of Σ^{-1}. Note that in 4.7, $(g'_{x_p})_{k_1-l_1,\cdots,k_p-1-l_p,\cdots,k_n-l_n}$ is essentially a derivative of g'_{x_p}, so

- If $\sum(k_i - l_i) > 1, (g'_{x_p})_{k_1-l_1\cdots k_n-l_n} = (-\underline{t}'_p\underline{x})_{k_1-l_1\cdots k_n-l_n} = 0$ because higher than 1st order partial derivatives of a linear expression will give 0.

- If $k_q - l_q = 1$, and other $k_j - l_j$ are all 0, $(g'_{x_p})_{k_1-l_1\cdots k_n-l_n} = (-\underline{t}'_p\underline{x})'_{x_q} = -t_{qp}$, and

- If all $k_i - l_i$ are 0, that is just $g'_{x_p} = -\underline{t}'_p \underline{x}$.

Hence there are at most $n + 1$ terms in 4.7. One is obtained by setting all the l_i's equal to their upper bound, which requires an (f) value from $k - 1$ family. The other n terms are obtained by setting one of the l_i's to 1 less than its upper bound while keeping the other l_i's at their upper bounds. This requires (f) values from $k-2$ family. All the other terms in 4.7 will vanish because they involve higher than first order partial derivatives of g'_{x_p}.

As in the trivariate Normal case, we can replace everything in 4.7 by its corresponding interval element. The resulting interval will be an interval inclusion of $(f)_{k_1 \cdots k_n}$ as is guaranteed in Section 2.3.1, because all the mathematical operations in 4.7 are rational.

4.1.4 The Use of MasPar

We used the 16384-processor MP1 for this integration problem. The computation is basically dealing with the n-variate case of 4.4. We set an m value, use 4.7 to compute interval inclusions of all the partial derivatives needed, then use 4.4 to obtain an interval inclusion of I. We then check whether or not the length of the inclusion is small enough. If smaller than our pre-specified tolerance the program will stop, otherwise we will raise m and start again.

The most computationally intensive part is the computation of all the partial derivatives up to a certain order m. Notice that for the rule part of the Taylor expansion 4.4 every time we increase m, we add some more terms to the previous expression. Hence the already computed partial derivatives of order lower than m do not have to be kept. Also for the remainder part in 4.4 every new m will result in a new

remainder which does not involve previously computed lower order partial derivatives. Therefore, for both derivatives evaluated at $(C_1, C_2, \cdots, C_n)$ and $(X_1, X_2, \cdots, X_n)$, we will focus on how to efficiently compute all the derivatives of an integer order k. Once this is obtained, 4.4 can be dealt with very easily.

Our computing method is an iterative procedure. We compute k family elements based on $k-1$ and $k-2$ family elements. Notice that the elements of the same k family do not depend on each other. Hence they can be assigned to different processors and computed simultaneously. We need the following corollary to determine how many elements are in a certain k family:

Corollary 4.2: Choose k from n different elements with replacement, the total number of combinations is $\binom{n+k-1}{k}$.

For a proof, see Lu (1983).

For an n-variate function f, the k-th order partial derivatives are in the form of $\frac{\partial^k}{\partial x_1^{k_1} \partial x_2^{k_2} \cdots \partial x_n^{k_n}} f$ with $k_1 + k_2 + \cdots + k_n = k$. All the derivatives are obtained by splitting k into n parts, or equivalently, choosing k from n different categories $(x_1, x_2, \cdots, x_n)$. From corollary 2, the total number of elements in k family is $\binom{n+k-1}{k}$.

Now we can assign the first PE on MP1 to compute the 0 family element, which is f itself; then we can assign the next $\binom{n+1-1}{1} = n$ PE's to let each of them compute an element from 1 family simultaneously; after that we assign the next $\binom{n+2-1}{2} = (n+1)n/2$ PE's to compute 2 family elements simultaneously, and so on. Figure 4.1 shows the layout of partial derivatives on the PE Array in the $n = 3$ case. But in detail, how does each assigned PE know which k family element it is computing? We developed the following order of k family elements for easy allocation of them on the $\binom{n+k-1}{k}$ PE's.

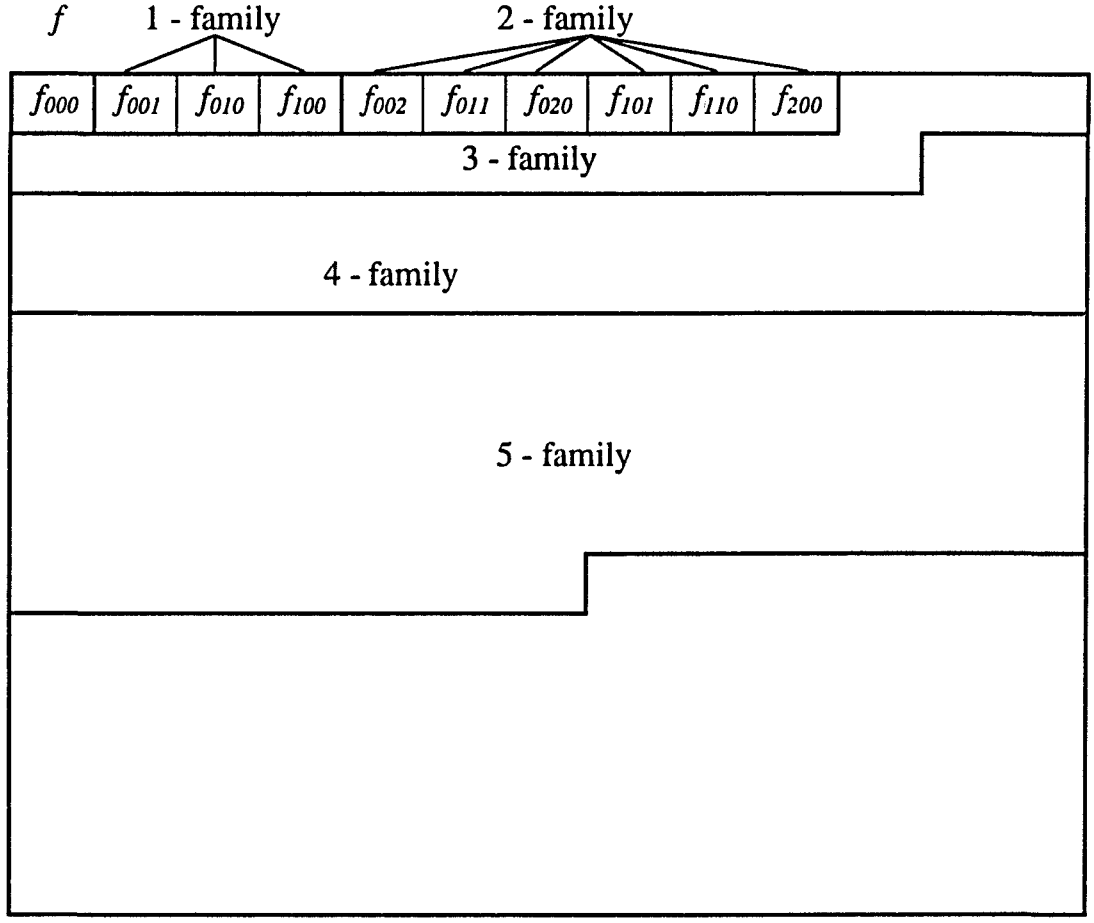

Figure 4.1: The layout of partial derivatives on the PE Array in the $n = 3$ case.

Definition 4.1: An n-tuple $(k_1, k_2, \cdots, k_n)$ is said to be before $(l_1, l_2, \cdots, l_n)$ if there exists a $p \in \{1, 2, \cdots, n\}$ such that $k_p < l_p$ and for $i < p, k_i = l_i$.

Definition 4.2: $(l_1, l_2, \cdots, l_n)$ is said to be behind $(k_1, k_2, \cdots, k_n)$ if $(k_1, k_2, \cdots, k_n)$ is before $(l_1, l_2, \cdots, l_n)$.

For example, in the case of $n = 3$ and $k = 4$, (0 0 4) is before (1 1 2), which is again before (1 3 0). A complete, ordered list for $n = 3$ and $k = 4$ is in Table 4.1.

Table 4.1: Complete ordered list of subscripts for the partial derivatives in the case of $n = 3$ and $k = 4$.

Order #	$(k_1 k_2 k_3)$
1	(0 0 4)
2	(0 1 3)
3	(0 2 2)
4	(0 3 1)
5	(0 4 0)
6	(1 0 3)
7	(1 1 2)
8	(1 2 1)
9	(1 3 0)
10	(2 0 2)
11	(2 1 1)
12	(2 2 0)
13	(3 0 1)
14	(3 1 0)
15	(4 0 0)

On the PE Array, for $n = 3$, in the PE area for computing 4 family elements we have $\binom{3+4-1}{4} = 15$ PE's and each PE will be assigned a 3-tuple (k_1, k_2, k_3) according to its order in this PE area and the list in Table 4.1. This will give us a mapping of all the partial derivatives of f onto the PE Array.

This mapping will help us in both finding the partial derivative to be computed on a particular PE and locating the PE that contains a needed lower family partial

derivative. In order to actually use the order table, we need a systematic way of performing the mapping.

From the PE order number to find $(k_1, k_2, \cdots, k_n)$, the partial derivative to be computed, we use a big loop statement in the program, not an explicit expression. In the $n = 3$ case, the loop looks like

```
pid = 0;
for (k1 = 0; k1 <= k; k1 ++)
for (k2 = 0; k2 <= k - k1; k2 ++)
  { pid ++;
    if (pid == id) k3 = k - k1 - k2;
  }
```

This is run on every PE processor to create the particular 3-tuple (k_1, k_2, k_3).

Since the computation of partial derivatives requires elements from $k-1$ and $k-2$ families, and according to our previous discussion, the n-tuples for them are already known at this point, we need to find ways to locate them on the PE Array. This is equivalent to finding the order number in the k family from n, k and $(k_1, k_2, \cdots, k_n)$. Let's consider an explicit form. For convenience, we will consider the number of terms in the same k family that are behind an n-tuple $(k_1, k_2, \cdots, k_n)$.

Each n-tuple is corresponding to a term in the expression of $(x_1 + x_2 + \cdots + x_n)^k$. Since

$$\begin{aligned}(x_1 + x_2 + \cdots + x_n)^k &= [x_1 + (x_2 + x_3 + \cdots + x_n)]^k \\ &= \sum_{a=0}^{k} \binom{k}{a} x_1^a (x_2 + x_3 + \cdots + x_n)^{k-a},\end{aligned}$$

all the n-tuples corresponding to $a > k_1$ are behind $(k_1, k_2, \cdots, k_n)$. The number of

such terms can be calculated as follows:

$$\begin{aligned}
a = k : \binom{n-1+0-1}{0} &= \binom{n-2}{0} \\
a = k-1 : \binom{n-1+1-1}{1} &= \binom{n-2+1}{1} \\
a = k-2 : \binom{n-1+2-1}{2} &= \binom{n-2+2}{2} \\
&\vdots \\
a = k_1+1 : \binom{n-1+k-(k_1+1)-1}{k-(k_1+1)} &= \binom{n-2+k-(k_1+1)}{k-(k_1+1)} \\
\text{Total} \quad : \quad & \binom{n-1+k-(k_1+1)}{k-(k_1+1)} \qquad (4.8)
\end{aligned}$$

Fix $a = k_1$, and consider $(x_2 + x_3 + \cdots + x_n)^{k-k_1}$. Similarly

$$\begin{aligned}
(x_2 + x_3 + \cdots + x_n)^{k-k_1} &= [x_2 + (x_3 + x_4 + \cdots + x_n)]^{k-k_1} \\
&= \sum_{b=0}^{k-k_1} \binom{k-k_1}{b} x_2^b (x_3 + x_4 + \cdots + x_n)^{k-k_1-b}.
\end{aligned}$$

All the n-tuples corresponding to $a = k_1$ and $b > k_2$, in addition to those corresponding to $a > k_1$, are also behind $(k_1, k_2, \cdots, k_n)$. The number of them can be calculated the same way as above with n and k replaced by $n-1$ and $k-k_1$, respectively, giving the total number

$$\binom{n-2+k-k_1-(k_2+1)}{k-k_1-(k_2+1)}.$$

Similarly, fix $a = k_1, b = k_2$, and so on. Finally, when the first $n-2$ elements of the n-tuple are all fixed, consider

$$\begin{aligned}
(x_{n-1} + x_n)^{k-k_1-\cdots-k_{n-2}} &= (x_{n-1} + x_n)^{k_{n-1}+k_n} \\
&= \sum_{c=0}^{k_{n-1}+k_n} \binom{k_{n-1}+k_n}{c} x_{n-1}^c x_n^{k_{n-1}+k_n-c}.
\end{aligned}$$

n-tuples corresponding to $c > k_{n-1}$ are all behind $(k_1, k_2, \cdots, k_n)$. Similarly as in 4.8, with n and k replaces by 2 and $k_{n-1} + k_n$, respectively, the total number for n-tuples for $c > k_{n-1}$ is

$$\binom{1 + k_{n-1} + k_n - (k_{n-1} + 1)}{k_{n-1} + k_n - (k_{n-1} + 1)} = \binom{k_n}{k_{n-1}} = \binom{k_n}{1}.$$

Summing them, we get the total number of terms that are behind $(k_1, k_2, \cdots, k_n)$:

$$\binom{n + k - k_1 - 2}{k - k_1 - 1} + \binom{n + k - k_1 - k_2 - 3}{k - k_1 - k_2 - 1} + \binom{n + k - k_1 - k_2 - k_3 - 4}{k - k_1 - k_2 - k_3 - 1} + \cdots + \binom{k_n}{1}$$

$$=$$

$$\binom{n + k - k_1 - 2}{n - 1} + \binom{n + k - k_1 - k_2 - 3}{n - 2} + \binom{n + k - k_1 - k_2 - k_3 - 4}{n - 3} + \cdots + \binom{k_n}{1}. \tag{4.9}$$

Here it is understood that $\binom{l}{m} = 0$ if $l < m$.

From Section 4.1.3 we know that to compute a partial derivative we need a value from the $k-1$ family and n values from the $k-2$ family. To actually fetch them we need to find their order numbers in their families. For the $k-1$ family member, we first get the n-tuple as described in Section 4.1.3, and then compute all the terms in 4.9. For the $k-2$ family members, since the n-tuples for them can be obtained by subtracting one from each of the n elements of the n-tuple for the $k-1$ family value, we can reduce each of the k_i's by one to modify the terms in 4.9 in order to obtain the orders for the $k-2$ family values. Notice that reducing k to $k-1$ and k_i to $k_i - 1$ for some i will not affect the last $n - 1 - i$ terms in 4.9, and the first i terms in 4.9 can be updated by multiplying by a factor. This will give us locations for the needed lower family partial derivatives quickly.

For each k family, we already know that the total number of values to be computed is $\binom{n+k-1}{k}$. As k increases, $\binom{n+k-1}{k}$ increases rapidly. If we assign one value

to each PE, we will soon run out of the 16384 processors. To overcome the shortage of processors, we introduce the Virtual PE Array (VPEA). VPEA is obtained by wrapping around the real PE Array 50 times. Because each processor has its own memory space, we can designate a 50-element array on each PE and let each of the element hold a partial derivative value. In addition, Since only the $k-1$ and $k-2$ family values have to be kept in order to compute k family values, those lower than $k-2$ family values can be overwritten, making room for more partial derivatives to be computed. Eventually, when even the VPEA is too small, we will use a recursive program to compute $(f)_{k_1 \cdots k_n}$ values. This actually uses the stack space on each PE. For different n this occurs at different times. Table 4.2 is a list of the biggest k family we can deal with before we go into the recursive part of the program.

Table 4.2: Maximum number of terms

n	$maxk$
4	116
5	49
6	29
7	21
8	17
9	14
10	12

Overall, the program works like this: we try to assign each $(f)_{k_1 \cdots k_n}$ to be computed on a PE node. When we come to the end of the PE Array, we can wrap around to the top of the PE Array and check whether or not the $(f)_{k_1 \cdots k_n}$ value there belongs to $k-1$ or $k-2$ families. If not, then we just treat it as empty and overwrite it; if yes, we will use the next element of the 50-element array on that PE to hold the $(f)_{k_1 \cdots k_n}$ value. This way we are using the VPEA, If even the 50-element array is

full, we will use the stack memory and start the recursive algorithm. In the program, we have to keep track of the head and tail of the VPEA, and head and tail of the 50-element array on each PE.

4.1.5 Performances

The program reads in dimension n, interval lower and upper bounds on each dimension, a_i, b_i, and upper triangular elements of Σ^{-1}, $t_{ij} (i \leq j)$. Output is an interval inclusion of the integration over the finite region. The current program can deal with a single hyper-cube with length on each dimension at most 1.9, ie, $b_i - a_i$ has to be ≤ 1.9 for all i. The number 1.9 is chosen so that after each iteration we get tighter inclusion. If the length is bigger than 1.9, we observed that the first few iterations get wider and wider inclusions, and it will begin to shrink after a few steps. This is caused by the remainder in the Taylor expansion. When n is small the remainder is dominated by the numerator. This restriction can be avoided by changing the program to a subroutine and dividing the whole region into several size "1.9" pieces with one call to the subroutine for each piece.

The recursive part of the program runs very slowly after we need more than $maxk$ terms in Table 4.2. This is as expected. For example,

- for $n = 10$, it takes 2.5 hours to compute all the 500,000 partial derivatives for a k family,
- for $n = 7$, it takes 1.5 hours to compute all the 376,000 partial derivatives for a k family,
- for $n = 8$, it takes 50 minutes to get results for $k = 17$. After 9 hours the

program is still on its way to the $k = 20$ results.

The non-recursive part and the recursive part shrink the interval inclusion at about the same speed. This should be true because it is only in the way they store the data that they are different.

If we restrict our attention to $k \leq maxk$, Table 4.3 is a summary of the accuracy for Multivariate Normal integration.

Table 4.3: The accuracy of multivariate Normal integration.

	Integration Interval on Each Dimension			
n	± 0.5	± 0.4	± 0.3	± 0.2
10	2 sig, 6 dp	3 sig, 8 dp	6 sig, 12 dp	10 sig, 17 dp
9	4 sig, 7 dp	6 sig, 10 dp	8 sig, 13 dp	11 sig, 18 dp
8	5 sig, 8 dp	7 sig, 11 dp	9 sig, 14 dp	13 sig, 19 dp
7	9 sig, 11 dp	12 sig, 15 dp		
6	13 sig, 15 dp			
5	13 sig, 15 dp			

sig: significant digits.
dp: decimal places.

We programmed the same algorithm on a DECstation 5000/240, which is a serial computer. Table 4.4 is a summary of the comparison of the running time between MP1 and the workstation.

Because the DECstation does not have as much memory as the MP1, it can deal with fewer terms before the recursive part starts. But we can see from Table 4.4, even for these fewer terms the running time is already much more than MP1. For example, in the $n = 10$ case MP1 deals with 13 terms, DECstation deals with only 9 terms. But the running time is already 50 minutes vs. 130 minutes. Note that the number of partial derivatives, which is approximately proportional to the running

Table 4.4: The comparison of the running time for multivariate Normal integration between MP1 and the workstation.

	MP1		DEC 5000	
n	# of terms	Running Time	# of terms	Running Time
10	13	50 min	9	130 min
9	15	60 min	10	187 min
8	17	50 min	11	150 min
7	22	75 min	14	309 min
6	31	110 min	18	360 min
5		18 min	28	> 10 hr

time, increases dramatically with increasing k. If the DECstation had the memory space to deal with 12 terms, we could imagine the running time to be 50 or 60 hours. Hence, the use of MP1 is really a significant speedup.

4.2 Multivariate t Distribution

In this section we want to develop a method to obtain interval inclusion for a Multivariate t probability. We choose the most commonly used definition of Multivariate t distribution, which can be found in Tong (1990). Suppose vector $\underline{Z} = (Z_1, Z_2, \cdots, Z_n)' \sim N(\underline{0}, R)$ and $\nu S^2 \sim \chi^2(\nu)$, then $\underline{t} = (t_1, t_2, \cdots, t_n)' = (Z_1/S, Z_2/S, \cdots, Z_n/S)' \sim t$ with ν degrees of freedom. Under this definition, if R is positive definite, the density of $\underline{t}$ is

$$h(\underline{t}; R, \nu) = \frac{\Gamma\left(\frac{n+\nu}{2}\right)}{(\nu\pi)^{n/2}\,\Gamma\left(\frac{\nu}{2}\right)|R|^{1/2}}\left(1 + \frac{1}{\nu}\underline{t}'R^{-1}\underline{t}\right)^{-\frac{n+\nu}{2}}, \quad \underline{t} \in \mathcal{R}^n.$$

Let

$$\begin{aligned} k_0 &= \frac{\Gamma\left(\frac{n+\nu}{2}\right)}{(\nu\pi)^{n/2}\,\Gamma\left(\frac{\nu}{2}\right)|R|^{1/2}}, \\ g(\underline{t}) &= 1 + \frac{1}{\nu}\underline{t}'R^{-1}\underline{t}, \qquad \text{and} \end{aligned}$$

$$f(\underline{t}) \;=\; (g(\underline{t}))^{-\frac{n+\nu}{2}}, \tag{4.10}$$

then

$$h(\underline{t}; R, \nu) = k_0 f(\underline{t}).$$

What we want to compute is

$$\begin{aligned} I &= P(a_1 < t_1 < b_1, a_2 < t_2 < b_2, \cdots, a_n < t_n < b_n) \\ &= \int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} h(\underline{t}; R, \nu)\, d\underline{t} \\ &= k_0 \int_{a_1}^{b_1}\int_{a_2}^{b_2}\cdots\int_{a_n}^{b_n} f(\underline{t})\, d\underline{t}. \end{aligned}$$

As in the case of the Multivariate Normal distribution, let

$$c_i = \frac{a_i + b_i}{2}, \qquad X_i = [a_i, b_i],$$

and expand $f(\underline{t})$ to its Taylor expansion at $\underline{c} = (c_1, c_2, \cdots, c_n)$,

$$\begin{aligned} f(\underline{t}) \;=\; & \sum_{k=0}^{m-1}\left[\sum_{\sqsupset k \sqsubset} \frac{1}{k_1!\cdots k_n!}\frac{\partial^k}{\partial t_1{}^{k_1}\cdots\partial x_n{}^{k_n}} f(\underline{c}) \prod_{i=1}^{n}(t_i - c_i)^{k_i}\right] \\ & + \sum_{\sqsupset m \sqsubset} \frac{1}{m_1!\cdots m_n!}\frac{\partial^m}{\partial t_1{}^{m_1}\cdots\partial t_n{}^{m_n}} f(\underline{\xi}) \prod_{i=1}^{n}(t_i - c_i)^{m_i}, \end{aligned}$$

where $\underline{\xi} = (\xi_1, \xi_2, \cdots, \xi_n)$ and $\xi_i \in X_i$. Consequently,

$$\begin{aligned} f(\underline{t}) \;\in\; & \sum_{k=0}^{m-1}\left[\sum_{\sqsupset k \sqsubset} \frac{1}{k_1!\cdots k_n!}\frac{\partial^k}{\partial t_1{}^{k_1}\cdots\partial x_n{}^{k_n}} f(\underline{c}) \prod_{i=1}^{n}(t_i - c_i)^{k_i}\right] \\ & + \sum_{\sqsupset m \sqsubset} \frac{1}{m_1!\cdots m_n!}\frac{\partial^m}{\partial t_1{}^{m_1}\cdots\partial t_n{}^{m_n}} f(\underline{X}) \prod_{i=1}^{n}(t_i - c_i)^{m_i}. \end{aligned}$$

Integrate according to the technique of Section 2.4, and we find that

$$I \in K_0 \cdot 2^n \sum_{\substack{k=0 \\ even}}^{m-1} \sum_{\substack{\sqsupset k \sqsubset \\ k_i even}} \frac{1}{(k_1+1)!\cdots(k_n+1)!}\frac{\partial^k}{\partial t_1{}^{k_1}\cdots\partial t_n{}^{k_n}} F(\underline{C}) \prod_{i=1}^{n} H_i{}^{k_i+1} +$$

$$K_0 \sum_{\sqsupset m \sqsubset} \frac{1}{(m_1+1)!\cdots(m_n+1)!} \overset{n}{(} \cdots \overset{3\,2\,1}{(((} \frac{\partial^m}{\partial {t_1}^{m_1}\cdots\partial {t_n}^{m_n}} F(\underline{X})\, {H_1}^{m_1+1} - \frac{\partial^m}{\partial {t_1}^{m_1}\cdots\partial {t_n}^{m_n}} F(\underline{X})$$
$$(-H_1)^{m_1+1} \overset{1}{)}\, {H_2}^{m_2+1} - \overset{1\,1}{(}\,)\,(-H_2)^{m_2+1} \overset{2}{)}\, {H_3}^{m_3+1} - \overset{2\,2}{(}\,)\,(-H_3)^{m_3+1} \overset{3}{)} \cdots \overset{n}{)}. \qquad (4.11)$$

where $h_i = \frac{b_i - a_i}{2}$ and H_i is the corresponding interval variable. The computation is rational and we need to worry about only how to obtain interval inclusions of the partial derivatives.

Let's use the same notations as in the Normal case. We want to compute $(f)_{k_1 k_2 \cdots k_n}$. Suppose $k_1 \geq 1$, then from 4.10,

$$f'_{t_1} = -\frac{n+\nu}{2} g^{-\frac{n+\nu}{2}-1} \cdot g'_{t_1}.$$

Hence

$$g \cdot f'_{t_1} = -\frac{n+\nu}{2} f \cdot g'_{t_1}$$

and

$$\frac{1}{k_1}\left(g \cdot f'_{t_1}\right)_{k_1-1,k_2,\cdots,k_n} = \frac{1}{k_1}\left(-\frac{n+\nu}{2}\right)\left(f \cdot g'_{t_1}\right)_{k_1-1,k_2,\cdots,k_n}.$$

Apply Generalized Leibniz's rule, we obtain

$$\begin{aligned}
LHS &= \frac{1}{k_1} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} \cdots \sum_{i_n=0}^{k_n} \left(f'_{t_1}\right)_{i_1 \cdots i_n} (g)_{k_1-1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1} \sum_{i_1=0}^{k_1-1} \sum_{i_2=0}^{k_2} \cdots \sum_{i_n=0}^{k_n} (1+i_1)(f)_{1+i_1,i_2,\cdots,i_n} (g)_{k_1-1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1} \sum_{i_1=1}^{k_1} \sum_{i_2=0}^{k_2} \cdots \sum_{i_n=0}^{k_n} i_1 (f)_{i_1 i_2 \cdots i_n} (g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1} \sum_{i_1=0}^{k_1} \sum_{i_2=0}^{k_2} \cdots \sum_{i_n=0}^{k_n} i_1 (f)_{i_1 i_2 \cdots i_n} (g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= (f)_{k_1 k_2 \cdots k_n} g + \frac{1}{k_1} \sum_{i_1=0}^{k_1} \sum_{\substack{i_2=0 \\ !(i_j=k_j,\forall j)}}^{k_2} \cdots \sum_{i_n=0}^{k_n} i_1 (f)_{i_1 i_2 \cdots i_n} (g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n}.
\end{aligned}$$

$$
\begin{aligned}
RHS &= \frac{1}{k_1}\left(-\frac{n+\nu}{2}\right)\sum_{i_1=0}^{k_1-1}\sum_{i_2=0}^{k_2}\cdots\sum_{i_n=0}^{k_n}(f)i_1 i_2\cdots i_n\left(g'_{t_1}\right)_{k_1-1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1}\left(-\frac{n+\nu}{2}\right)\sum_{i_1=0}^{k_1-1}\sum_{i_2=0}^{k_2}\cdots\sum_{i_n=0}^{k_n}(k_1-i_1)(f)_{i_1 i_2\cdots i_n}(g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1}\left(-\frac{n+\nu}{2}\right)\sum_{i_1=0}^{k_1}\sum_{i_2=0}^{k_2}\cdots\sum_{i_n=0}^{k_n}(k_1-i_1)(f)_{i_1 i_2\cdots i_n}(g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n} \\
&= \frac{1}{k_1}\left(-\frac{n+\nu}{2}\right)\sum_{i_1=0}^{k_1}\sum_{\substack{i_2=0 \\ !(i_j=k_j,\forall j)}}^{k_2}\cdots\sum_{i_n=0}^{k_n}(k_1-i_1)(f)_{i_1 i_2\cdots i_n}(g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n}.
\end{aligned}
$$

If we set $LHS = RHS$, we obtain

$$
(f)_{k_1 k_2\cdots k_n} =
$$

$$
\frac{1}{k_1 g}\sum_{i_1=0}^{k_1}\sum_{\substack{i_2=0 \\ !(i_j=k_j,\forall j)}}^{k_2}\cdots\sum_{i_n=0}^{k_n}\left[\left(-\frac{n+\nu}{2}\right)(k_1-i_1)-i_1\right](f)_{i_1 i_2\cdots i_n}(g)_{k_1-i_1,k_2-i_2,\cdots,k_n-i_n}. \tag{4.12}
$$

Again, as in the Normal case, $g(\underline{t})$ is a quadratic form. Hence higher than second order partial derivatives are all zero. To carry out the computation for a given n-tuple $(k_1, k_2, \cdots, k_n)$ in the k family, we can let one i_j be $k_j - 1$ (if this $k_j \neq 1$) and all the other i_j's be their corresponding k_j's. This gives us at most n elements from the $k-1$ family. Next let i_l and i_m be $k_l - 1$ and $k_m - 1$ respectively (if $k_l \neq 1$ and $k_m \neq 1$) while all the other i_j's take their corresponding k_j's. This will give us at most $\binom{n}{2} = n(n-1)/2$ elements from the $k-2$ family. Finally we let some i_j be $k_j - 2$ (if $k_j \geq 2$) and all other i_j's be the corresponding k_j's. This will give us at most n elements from $k-2$ family. Hence the total number of terms that contribute to the computation of $(f)_{k_1 k_2\cdots k_n}$ is at most

$$
n + \frac{n(n-1)}{2} + n = \frac{n^2+3n}{2},
$$

and 4.12 is sized down greatly.

If $k_1 = 0$ but $k_2 \neq 0$, or the other cases, 4.12 will be similar.

In 4.12 we can replace everything with corresponding interval elements to obtain an interval inclusion for $(f)_{k_1 k_2 \cdots k_n}$. This can consequently be used in 4.11 to compute an interval inclusion for the Multivariate t integral.

On MasPar, since for any k family the member of partial derivatives to be computed is exactly the same as the Normal case, the layout of the (f) values on the MP1 PE Array can be exactly the same as what is done in the Normal case. And the usage of the PE Array is in the same way. Table 4.5 is a table of running time and accuracy for the Multivariate t distribution on MP1. The degrees of freedom is $\nu = 10$.

Table 4.5: Running time and accuracy for the multivariate t distribution on MP1.

	MP1		Interval on Each Dimension				
n	# of Terms	Running Time	±0.5	±0.4	±0.3	±0.2	±0.1
10	12	80 min			2 sig 7 dp	4 sig 11 dp	10 sig 20 dp
9	14	70 min			3 sig 8 dp	7 sig 13 dp	
8	17	85 min		0 sig 3 dp	5 sig 9 dp	10 sig 15 dp	
7	21	90 min		3 sig 6 dp	8 sig 12 dp		
6	29	110 min	3 sig 5 dp	8 sig 10 dp			
5	49	3 hr	10 sig 11 dp				

sig: significant digits.
dp: decimal places.

Compared with Normal distribution, the running time is longer. This is because in the Normal case we need at most $n+1$ lower family (f) values to compute the current (f) value, while in the t case we need at most $(n^2+3n)/2$ lower family (f) values. Therefore more computation is involved. This is also a factor causing the computed interval inclusion wider than in the Normal case.

Another thing that is worth noting is that when the integration region is near the origin, the interval inclusion will be better when ν gets bigger; when the integration region is off the origin, the interval inclusion will be better when ν gets smaller. As shown in Figure 4.2, it looks like bigger integration values tend to get tighter inclusions. We think this is caused by the nature of the integration itself. For example, for integration regions near the origin, bigger ν will make the inclusion for $\left(1+\frac{1}{\nu}\underline{t}'\underline{t}\right)$ tighter, hence the whole integration will be tighter. A numeric example follows : If we set R^{-1} to be the identity matrix, the pdf becomes $\left(1+\frac{1}{\nu}\underline{t}'\underline{t}\right)^{-\frac{n+\nu}{2}}$. Then for the integration region with interval on each dimension $[-0.5, 0.5]$,

if $n=2, \nu=1$: $\quad 1+\frac{1}{\nu}\underline{t}'\underline{t} = 1+\frac{1}{\nu}(0,0.5) = (1,1.5),\ f(\underline{t}) = (0.544,1),$

if $n=2, \nu=10$: $\quad 1+\frac{1}{\nu}\underline{t}'\underline{t} = (1,1.05),\ f(\underline{t}) = (0.746,1).$

Hence the computation for $\nu=1$ gives a wider interval.

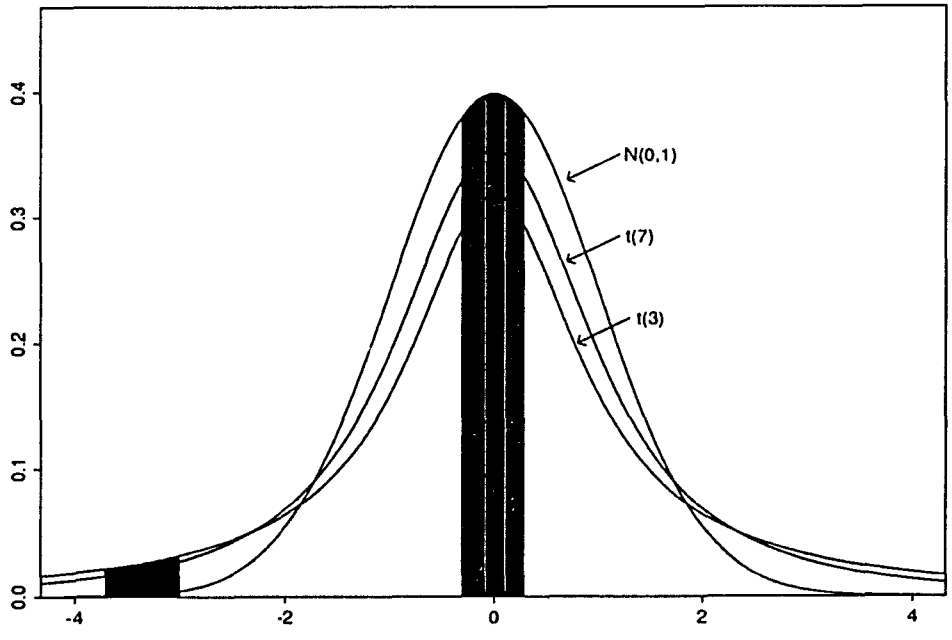

Figure 4.2: Normal and t distribution plots.

CHAPTER 5. OPTIMIZATION

In the previous chapter we used interval analysis mainly to bound computational errors. This is an important use of interval analysis and was the initial motivation for its development. But in subsequent research, other areas have been discovered where interval analysis can be of great help. One of them that has major importance for statistical analysis is global optimization.

The following is an incomplete list of the statistical problems that make use of optimization techniques :

1. Nonlinear Regression.

   The model function $f(\underline{\beta}, \underline{x})$ can be very complicated. Least squares estimation is commonly used. This involves determining the value or values of $\underline{\beta}$ to minimize

   $$\sum_{i=1}^{n} \left(y_i - f(\underline{\beta}, \underline{x_i})\right)^2.$$

   A popular minimization strategy approximates the nonlinear model function with a Taylor expansion, usually to only two terms, then minimize the residual sum of squares of the polynomial model using the standard method for linear regression. The result is used as the point at which to expand the nonlinear model function for the next step. We continue this process until it converges.

2. Maximum Likelihood Estimation:

   As in the regression case, the likelihood function can be very complicated. Using MLE in factor analysis, for example, can involve many parameters.

3. Optimal Experimental Design:

   It deals with the problem of how to choose different experimental points in order to estimate the model parameters with the greatest accuracy, in some sense. There are different criteria to define optimality, e.g., a D-optimal design minimizes $|X'X|^{-1}$, where X is the model matrix, an A-optimal design minimizes $\mathrm{tr}(X'X)^{-1}$, an E-optimal design minimizes the largest eigenvalue of $(X'X)^{-1}$, and a G-optimal design minimizes the maximum prediction variance $\max_{\underline{x}} Var(\hat{y}(\underline{x}))$. There are many other criteria.

4. Bayesian Theory:

   Optimization may arise in Bayesian estimation problems, e.g., the estimation of a parameter may require the mode of the posterior distribution. This means we have to maximize the posterior pdf, which is often complicated.

   In Bayesian decision theory, a Bayes rule minimizes the expected loss. This may be easy for certain special loss functions like squared error loss, but in general it is a difficult optimization problem.

5. Image Processing:

   We want to determine the true color or grey level on each pixel, based on the picture data. Solving this image processing problem means we define some objective function, taking into account information on each pixel, and then

minimize a loss function. A picture usually comprises many pixels, and we have to optimize with respect to many variables.

All the above problems require numerical global optimization. In Section 5.1 we will review commonly used optimization methods and the problems associated with them. In Section 5.2 we introduce, based on interval analysis, an optimization method on the MasPar machine that is guaranteed to find the global optimizer. Section 5.3 contains applications of this optimization method to several statistical problems.

5.1 Conventional Methods

We will discuss the steepest descent method, Newton's method and modifications, and simulated annealing. We will talk about how they work and their usefulness and shortcomings in various cases.

Steepest Descent Method

This is one of the oldest methods for determining the minimum of a function. It is generally accepted that Cauchy first proposed this method in 1847.

For a function $f(\underline{x})$, at any given point $\underline{x}_i$, the gradient vector

$$\underline{g}_i = \underline{g}(\underline{x}_i)$$

defines the direction of maximum local increase in $f(\underline{x})$. Thus if we want to locate the local minimum, we should proceed most rapidly downhill and consider the direction opposite to $\underline{g}_i$. This defines the iteration for steepest descent method :

$$\underline{x}_{i+1} = \underline{x}_i - \alpha_i \underline{g}_i.$$

Here α_i is the step size. For the choice of step size see Kennedy and Gentle (1980).

The steepest descent method often tends to make good progress initially and then slows down as it approaches a stationary point. Also, we have to be aware that it can be easily trapped by local minimum.

Newton's Method and Modifications

Newton's method is a more widely used and more often studied method. Consider the Taylor expansion of f at the point $\underline{x}_i$:

$$f(\underline{x}) = f(\underline{x}_i) + (\underline{x} - \underline{x}_i)'\underline{g}_i + \frac{1}{2}(\underline{x} - \underline{x}_i)'G_i(\underline{x} - \underline{x}_i) + E \tag{5.1}$$

where G_i is the Hessian matrix and E is the remainder. When $\underline{x}$ is close to $\underline{x}_i$ we expect the rule part of 5.1 to approximate $f(\underline{x})$ with very little error. We can then try to find the stationary point of 5.1, which is given by the solution to

$$G_i(\underline{x} - \underline{x}_i) = -\underline{g}_i.$$

If G_i is positive definite, the iteration becomes

$$\underline{x}_{i+1} = x_i - G_i^{-1}\underline{g}_i.$$

If G_i is not positive definite, many modifications to the Newton's method have been introduced. Some people suggest replacing all negative eigenvalues of G_i with their absolute values and replacing zero eigenvalues with a small positive number. This gives a positive definite G_i which can be used for the iteration. Another modification is to replace G_i with $G_i + \tau_i I_P$ where I_P is identity matrix and $\tau_i \geq 0$ is chosen so that $G_i + \tau_i I_P$ is positive definite. In case the Hessian matrix is difficult or impossible to obtain, various methods to approximate it have been suggested. The most famous of these are called Quasi-Newton methods. See Kennedy and Gentle (1980).

Newton's method and its modifications require user provided information G_i, and in return for this inconvenience the methods may converge to a stationary point at a quadratic rate. It is significantly more efficient than steepest descent method. The major drawback is that it still can not guarantee to find the global optimizer.

Simulated Annealing

The general descent iteration proceeds by taking a step from $\underline{x}_i$ to $\underline{x}_{i+1}$. The change in function value is

$$\triangle f = f(\underline{x}_{i+1}) - f(\underline{x}_i).$$

Simulated Annealing requires that the step from $\underline{x}_i$ to $\underline{x}_{i+1}$ be accepted with probability

$$p = \begin{cases} 1 & \triangle f \leq 0 \\ exp\{-\gamma(T)\,\triangle f\} & \triangle f > 0 \end{cases}$$

with given $\gamma(T) > 0$. Note that the method will, with nonzero probability, allow a step uphill when attempting to find a global minimum. Occasional acceptance of uphill steps is intended to avoid being trapped at a local minimum. In order that the sequence of iterations converges to the global minimum, the probability of accepting uphill moves should converge to zero as the search proceeds. This is achieved by a gradual increase of $\gamma(T)$. However, if $\gamma(T)$ is increased too fast the algorithm may again be trapped in a local minimum. But an increase slow enough to convergence to the global optimum can involve extremely long running time. We have to balance between these two choices.

Simulated Annealing gives us a means to overcome the local minima problem. It is reassuring in theory and it works well for some applications, but for other

applications it may not be practical. Since we never know what kind of $\gamma(T)$ is appropriate, we have to set a cut-off run time and this will mean that the computation can stop before finding the global minimum. Hence in practice, simulated annealing can not guarantee that the global optimum will be found.

In this section we discussed traditional optimization techniques. The common problem with all of them is that the algorithm may very well just give a local minimum, leaving the global minimum undiscovered. Here is an example from Seber and Wild (1989) as an illustration of this problem.

Example: We have the data set as in Table 5.1.

Table 5.1: Data set for the Seber and Wild (1989) example

x_i	y_i
-2	0
-1	1
1	-0.9
2	0

We want to fit the following model by means of least squares

$$y_i = \alpha e^{-\beta x_i} + \epsilon_i.$$

In other words, we want to minimize

$$SS(\alpha, \beta) = \sum_{i=1}^{4} \left(y_i - \alpha e^{-\beta x_i}\right)^2.$$

A contour plot of the error sum of squares within the region $-0.2 < \alpha < 0.3$, $-1.2 < \beta < 1.3$ is presented in Figure 5.1.

$M_1(\alpha = 0.087, \beta = 0.62)$ is a global minimum with error sum of squares 1.69. $M_2(\alpha = -0.063, \beta = -0.699)$ is a local minimum with error sum of squares 1.73.

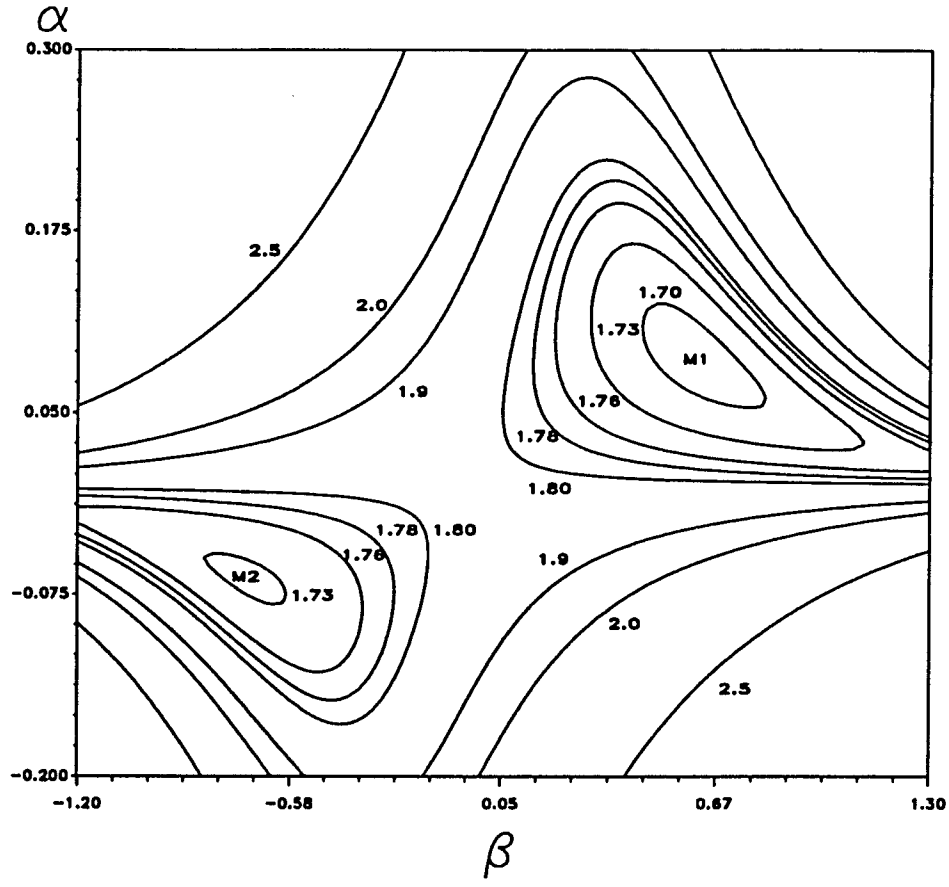

Figure 5.1: Contour plot for the Seber and Wild (1989) example.

Using the nonlinear regression routine in SAS, beginning at (0.3, 1.3) one obtains $\hat{\alpha} = 0.087, \hat{\beta} = 0.62$ and $SSE = 1.69$. But if the starting point $(-0.2, -1.2)$ is used one obtains $\hat{\alpha} = -0.063, \hat{\beta} = -0.699$ and $SSE = 1.727$. Hence when using standard methods, if we start from different initial points, we end up getting different results, sometimes the global minimum, sometimes a local minimum. We tried different method options in SAS *proc nlin*, and they all have the above problem.

In the rest of this chapter, we will develop an optimization method based on

interval analysis which provides guaranteed bounds on the globally optimal value f^* of an objective function f and on the point(s) x^* where it occurs.

5.2 Interval Optimization

5.2.1 Method

Interval analysis uses intervals as computing elements. From the discussion in Chapter 2, it has the capability of computing an inclusion for the range of a function over a certain region. Based on this, we develop our algorithm on a serial computer as follows. Because maximization problems and minimization problems are essentially the same, from now on we will just consider minimization.

For a given finite starting parameter region, we want to locate the minimum function point within that region. We can first split the whole region into two pieces, and then use interval analysis to obtain interval inclusions for the range of the objective function over the two pieces. Use the deciding procedures, which will be discussed in Section 5.2.3, to delete one of the pieces that is determined can not contain the global minimum. The deciding procedures use interval analysis in such a way that, despite rounding errors, the point(s) of global minimum is never deleted. For the remaining piece we can do the similar thing: divide it further into two subregions, use interval analysis and the deciding procedures to throw away one of them. We will keep doing this until the region that is left has the size smaller than a pre-specified tolerance. Then since the global minimum function point x^* must lie in this small region, its location is bounded to some pre-specified accuracy. Using interval analysis on this region we can get a bound for the global minimum function value. The whole process is illustrated in Figure 5.2, and the numbers on the figure denote the order in which

each piece is thrown away.

Notice that there is actually a restriction posted on the interval optimization algorithm. We always assume that the global optimum is contained in the finite starting region. If there is another point outside the region that possesses an objective function value that is smaller than any point in the region, the algorithm will not be able to find it. But this is not a very serious restriction because we can always start with a big enough region.

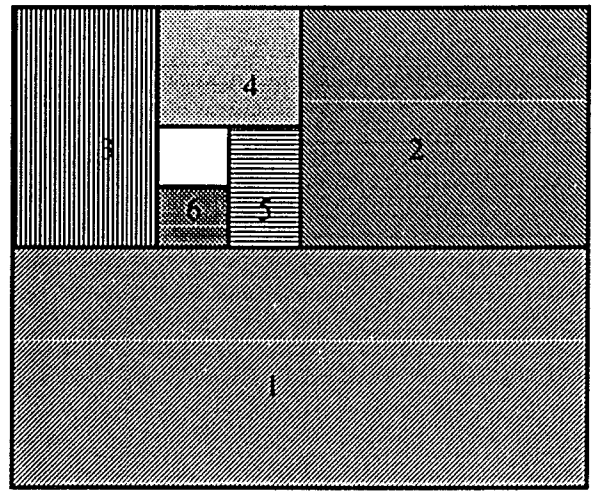

Figure 5.2: Interval optimization on a serial computer.

When splitting a region into two parts, we want to follow a strategy such that the next step of the algorithm can most easily throw away one of them. From our deciding procedures, this means we want the function values on them to be significantly different: one is always bigger than the other. An intuitive guess is to

split the dimension along which, while other variables are fixed, the function value varies the most. But the problem with this is that it is another optimization problem that would have to be computed for all n dimensions. Also, fixing other dimensions at different values will give different results. A relatively easier choice is to just split the dimension that has the biggest interval. But this proves to be too crude to meet the above requirements. So instead we follow the procedure described in Hansen (1992) to find another quantity that is a good indicator of how the function varies along a single dimension. It works as follows.

Suppose $X_i = [a_i, b_i]$ and define

$$f_i(t) = f(x_1, \cdots, x_{i-1}, t, x_{i+1}, \cdots, x_n),$$

and

$$g_{ii}(t) = g_i(x_1, \cdots, x_{i-1}, t, x_{i+1}, \cdots, x_n),$$

where g_i is the ith component of the gradient of f, x_i is the midpoint of the interval on dimension i. Also define

$$m_i = \min_{t \in X_i} g_{ii}(t), \quad M_i = \max_{t \in X_i} g_{ii}(t),$$

then

$$\max_{t \in X_i} f_i(t) \leq f_i(a_i) + \int_{a_i}^{b_i} M_i d\mu,$$

and

$$\min_{t \in X_i} f_i(t) \geq f_i(a_i) + \int_{a_i}^{b_i} m_i d\mu.$$

Therefore, since

$$M_i - m_i \leq \text{width}[g_{ii}(X_i)],$$

we have

$$\max_{t \in X_i} f_i(t) - \min_{t \in X_i} f_i(t) \leq \text{width}[g_{ii}(X_i)] \cdot \text{width}[X_i].$$

If we replace each x_j by X_j in the RHS, the relation is true for any $x_j \in X_j$ $(j = 1, \cdots, n, j \neq i)$ in the LHS. That is, for all such x_j,

$$\max_{t \in X_i} f_i(t) - \min_{t \in X_i} f_i(t) \leq \text{width}[g_{ii}(X)] \cdot \text{width}[X_i], \quad i = 1, 2, \cdots, n. \tag{5.2}$$

We can not say that the i for which the LHS is large will also be the one for which the RHS is large. Nevertheless, we expect good correlation. Hence we compute the RHS and find the i that makes it largest and cut the region into half on that dimension. This works fine compared with just cutting the dimension that has the biggest interval.

5.2.2 The Use of MasPar

We use MP2 for the interval optimization. On the MP2, since it has 4096 processors, we can take advantage of this and make the algorithm in Section 5.2.1 more efficient. Instead of bisecting, we can split the region into 4096 pieces and assign each one of them to a different processor. They will then run through the deciding procedures simultaneously to throw away most of the 4096 pieces. For the remaining pieces we can divide them again into 4096 even smaller regions and redistribute among the PE Array. Repeat this procedure until the regions that are left are all very small in size. Figure 5.3 illustrates the repeated process.

For the starting n-dimensional region, we need an algorithm to split it into 4096 pieces. Since this is the beginning of the algorithm, we do not have the derivative

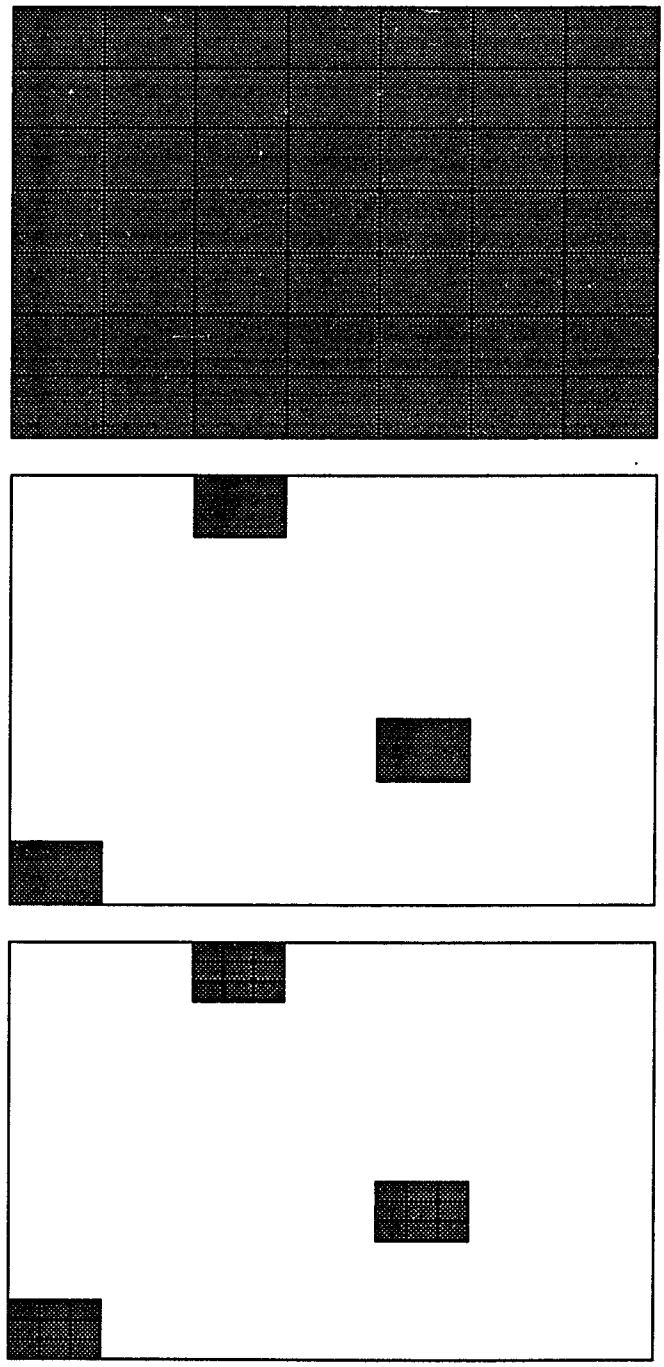

Figure 5.3: The change of MP2 PE Array during interval optimization.

information yet that we can use in the way mentioned in Section 5.2.1 to make the cuttings most beneficial, we will just tend to make the interval on each dimension as close to each other in size as possible. Hence a dimension with shorter length is always cut after a dimension with bigger length.

Every time we make the cut the interval is always cut into two pieces with equal length. we do this 12 times because $4096 = 2^{12}$, and after 12 times of cutting we have 4096 subregions. Table 5.2 shows an example of the 12 cuttings for an initial 3-dimensional region with length 10, 3, 2 on each dimension.

Table 5.2: Example of the 12 cuttings for a 3-dimensional region with length 10, 3, 2 on each dimension.

Cut #	D1	D2	D3
0	10	3	2
1	5	3	2
2	2.5	3	2
3	2.5	1.5	2
4	1.25	1.5	2
5	1.25	1.5	1
6	1.25	0.75	1
7	0.625	0.75	1
8	0.625	0.75	0.5
9	0.625	0.375	0.5
10	0.3125	0.375	0.5
11	0.3125	0.375	0.25
12	0.3125	0.1875	0.25

The cuttings are done on ACU, the non-parallel part of MP2. We keep track of the number of cuttings on each dimension $C_i, i = 1, 2, \cdots, n$ and the final length of the pieces on each dimension $P_i, i = 1, 2, \cdots, n$. After this we want to distribute the pieces to the PE Array. This is done in parallel and we developed an algorithm to let each PE actually compute out what subregion it should have. Each processor has a

PE number associated with it called *iproc*. Its range is from 0 to 4095. We perform the following computations on each PE:

$$
\begin{aligned}
iproc &= q_n C_n + r_n, && 0 \le r_n < C_n; \\
q_n &= q_{n-1} C_{n-1} + r_{n-1}, && 0 \le r_{n-1} < C_{n-1}; \\
&\cdot && \cdot \\
&\cdot && \cdot \\
&\cdot && \cdot \\
q_3 &= q_2 C_2 + r_2, && 0 \le r_2 < C2; \\
q_2 &= q_1 C_1 + r_1, && 0 \le r_1 < C_1; \qquad (5.3)
\end{aligned}
$$

and use the r_1th of the C_1 pieces of dimension 1, the r_2th of the C_2 pieces of dimension $2, \cdots$, and r_nth of the C_n pieces of dimension n as the subregion to be processed on each PE. We have 4096 pieces of subregions and 4096 processors, and we will prove that there is a one-on-one relation between them if we use the above strategy to distribute the subregions. In other words, each subregion will be assigned to a PE, and no two subregions are assigned to the same PE. This is equivalent to prove the following:

Corollary 5.1: In 5.3 let $C_i, i = 1, \cdots, n$ be all fixed, then *iproc* uniquely determines the n-tuple $(r_1, r_2, \cdots, r_n)$; and vise versa.

Proof: From the division algorithm (Rosen, 1984), given *iproc* and C_n, q_n and r_n are uniquely determined, then given q_n and c_n, q_{n-1} and r_{n-1} are uniquely determined, and so on. Hence the first part is proved.

For the second part, we can combine all the equations in 5.3 to obtain

$$iproc = q_n C_n + r_n$$

$$
\begin{aligned}
&= (q_{n-1}C_{n-1} + r_{n-1})C_n + r_n \\
&= q_{n-1}C_{n-1}C_n + r_{n-1}C_n + r_n \\
&= (q_{n-2}C_{n-2} + r_{n-2})C_{n-1}C_n + r_{n-1}C_n + r_n \\
&= q_{n-2}C_{n-2}C_{n-1}C_n + r_{n-2}C_{n-1}C_n + r_{n-1}C_n + r_n \\
&= \cdots \\
&= q_1C_1C_2\cdots C_n + r_1C_2\cdots C_n + r_2C_3\cdots C_n + \cdots + r_{n-1}C_n + r_n \\
&= q_1 \cdot 4096 + r_1C_2\cdots C_n + r_2C_3\cdots C_n + \cdots + r_{n-1}C_n + r_n,
\end{aligned}
$$

since $C_1 \cdots C_n = 4096$. Note that $0 \leq iproc < 4096$ and everything in 5.3 is nonnegative, this forces q_1 to be 0 and

$$iproc = r_1C_2\cdots C_n + r_2C_3\cdots C_n + \cdots + r_{n-1}C_n + r_n.$$

Hence given $C_i, i = 1, 2, \cdots, n$, $iproc$ is uniquely determined by the n-tuple $(r_1, \cdots, r_n)$. □

In the program, 5.3 is performed simultaneously on each PE to get $(r_1, \cdots, r_n)$. Together with the already obtained $P_i, i = 1, \cdots, n$, all the PEs can get the exact subregion assigned to them.

The above approach is for the initial splitting. Once we have completed an iteration, suppose most of the regions are thrown away, we want to more efficiently split the remaining regions and distribute them to the PE's with thrown away regions. We call the PEs with thrown away regions "cleared nodes", otherwise "uncleared nodes". The strategy is as follows:

We first mark all the uncleared nodes. Then all the uncleared nodes look at their right neighbor one-unit distance away. If it is cleared, the node will split its region into two pieces according to the method described in Section 5.2.1, put one of them

to this right neighbor and mark it uncleared; otherwise no action is taken. Then the nodes will look at their right neighbor two-unit distance away, repeat the above, and then three-unit distance away, and so on, until all the PE's are marked uncleared. Notice that all the uncleared nodes are working in parallel, so the splitting process is working very rapidly.

In real programming, to assure those "worst" regions to be split, we first count the number of cleared nodes and uncleared nodes. If the number of cleared nodes is greater than half of the total number of PE, we just proceed with the above strategy. Otherwise we order the "badness" of the uncleared nodes according to the RHS of 5.2. Only the first few that is equal in number to the cleared nodes will be split. The other "not so bad" ones will be marked specially and left untouched. This helps in reducing the number of remaining regions after each iteration.

Another implementation detail is that when the tolerance requirement is met, usually there is more than one subregion to output. Some of them may be bounding multiple global optima. Very often many of them are just bounding the points that are too close in function values to the global minimum point that they are not thrown away yet. Figure 5.4 shows an example of the final stage of an interval optimization problem.

All the shaded regions are still remained, and the global minima are in these areas. From the figure it is reasonable to say that there are two global optimal points, one in the region $[0.1, 0.3] \times [0, 0.3]$ and the other in $[0.9, 1.1] \times [0.2, 0.4]$. If we just output all the remaining regions there will be 7 scattered regions instead of 2 informative regions each bounding a global minimum point. Hence we want to develop a systematic way of grouping close enough subregions and make the output

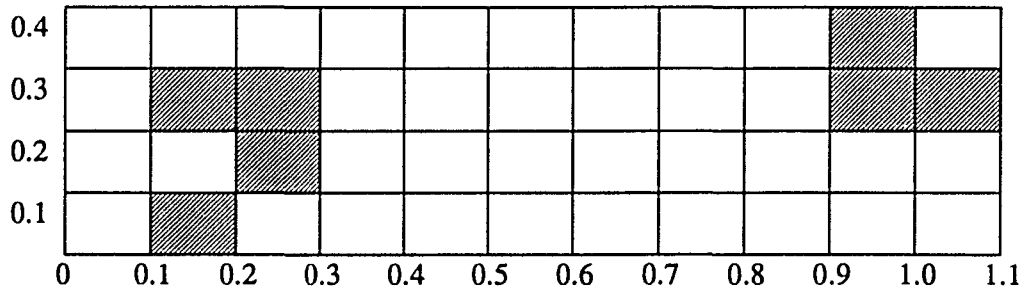

Figure 5.4: The regions that are left when it is ready to output.

more meaningful.

First we notice that all the subregions are disjoint. This is because at the beginning we split the whole region into 4096 non-overlapping regions, and at later steps a region is split into half which will never generate overlapping subregions. Disjoint regions have disjoint intervals on every dimension. We define the distance between two disjoint intervals $[a, b]$ and $[c, d]$ to be $c - b$. Then we can start grouping regions by dimension. On each dimension, first we sort all the intervals of the regions on this dimension by position. We then find the biggest interval length and multiply it by a pre-specified constant to get the distance criterion. If the distance between two intervals is smaller than this criterion they are considered to be in the same group. We assign this relation to be transitive, i.e., if intervals A and B are in the same group, B and C are in the same group, then A, B, C are all in the same group. we do this grouping on every dimension. After this the n-dimensional space is split into small pieces by the grouping on single dimensions. And we will group subregions $R_1, R_2, \cdots, R_k$ together if and only if they are in the same group on every dimension.

Suppose on dimension i the intervals are $R_{1i}, \cdots, R_{ki}$, then instead of outputting k regions $R_1, \cdots, R_k$, we output a single region with interval on dimension i being

$$\left[\min\left(\underline{R_{1i}}, \cdots, \underline{R_{ki}}\right), \max\left(\overline{R_{1i}}, \cdots, \overline{R_{ki}}\right)\right].$$

For example, in Figure 5.4, instead of outputting 7 regions, we group them together and output only 2 regions

$$[0.1, 0.3] \times [0, 0.3] \text{ and } [0.9, 1.1] \times [0.2, 0.4].$$

5.2.3 Deciding Procedures

As mentioned in Section 5.2.1, we use some deciding procedures to determine which regions can not contain the global optimum. Now we discuss them in detail.

We have 3 criteria available, and each subregion has to pass all of them in order to be retained.

1. A "function value" Criterion: Suppose the whole region is divided into smaller pieces, as shown in Figure 5.5.

   We then perform a random sampling on each of the subregions and compute the function values on the sample points. When we evaluate $f(x)$, we use interval arithmetic to bound rounding errors. Thus for every sample point we obtain an interval result $[\underline{f(x)}, \overline{f(x)}]$. Despite rounding errors, we know without question that $\overline{f(x)}$ is an upper bound for $f(x)$ and hence for the global minimum function value f^*. Let $\overline{f}$ denote the smallest such bound obtained for the sample points, we have

$$f^* \leq \overline{f}. \tag{5.4}$$

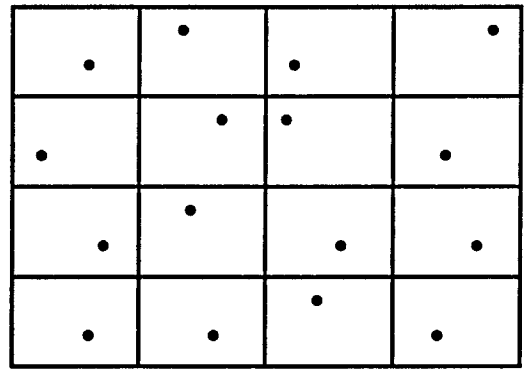

Figure 5.5: Splitting the region for deciding procedure #1.

Now let X be one of the subregions. We can use interval analysis to compute an interval inclusion $[\underline{f(X)}, \overline{f(X)}]$ for the range of f over X. Then, it is certain that $\underline{f(X)}$ is a lower bound for $f(x)$ for any $x \in X$. Hence, if $\underline{f(X)} > \overline{f}$, then from 5.4, $\underline{f(X)} > f^*$ and $f(x) > f^*$ for every $x \in X$. Therefore no point in the subregion X can be the global minimum point, and the region X can be thrown away.

2. A "derivative" Criterion: For most of the models used in statistics, the objective function is differentiable. Hence we incorporate derivative information into our algorithm. This information is also required in further splitting regions as shown in Section 5.2.1. We provide explicit forms of all the first order partial derivatives of the objective function, and use interval analysis to compute interval inclusions of the range of the derivatives over all the subregions. So for each dimension i, we will obtain an interval $[a_i, b_i]$ as an inclusion for the first order partial derivative with respect to variable number i. We know that at

the global minimum point all the first order partial derivatives are zero. Hence if for some region at least one of the inclusions for partial derivatives does not contain zero, the global minimum does not occur in it, so it can be thrown away. There is an exception to this rule : if a derivative does not contain zero but the region is on the original border of that dimension, it should be kept, because if the global minimum is on the border, the derivative need not be zero.

3. A "linearization" Criterion: The previous two criteria are of an "all or none" nature, in that a whole subregion is either kept or dropped from consideration. Sometimes we might just want to throw away part of a subregion. A criteria can be fashioned to allow this. We first linearize the function f, then bound all but one of the variables by their corresponding intervals. For $\underline{x}_0$, the middle point of the subregion, and $G_i(X_1, \cdots, X_n)$, the ith component of the interval inclusion for the partial derivatives of f,

$$\begin{aligned} f(\underline{x}) &\in f(\underline{x}_0) + \sum_{i=1}^{n} (x_i - x_{i0})\, G_i(X_1, \cdots, X_n) \\ &\in f(\underline{x}_0) + (x_j - x_{j0})\, G_j(X_1, \cdots, X_n) + \sum_{i \neq j} (X_i - x_{i0})\, G_i(X_1, \cdots, X_n). \end{aligned} \tag{5.5}$$

The right hand side of 5.5 is an interval. If its lower endpoint is greater than $\overline{f}$, then it is greater than f^*, the global minimum can not occur in it and the whole subregion can be discarded. But if that is not the case, we let

$$f(\underline{x}_0) + (w_j - x_{j0})\, G_j(X_1, \cdots, X_n) + \sum_{i \neq j} (X_i - x_{i0})\, G_i(X_1, \cdots, X_n) > \overline{f} \tag{5.6}$$

and solve this interval inequality to find out which part of the region will make

the inequality true and throw away just that part. To solve it, define $t = w_j - x_{j0}$. Let

$$U = f(\underline{x}_0) + \sum_{i \neq j} (X_i - x_{i0})\, G_i(X_1, \cdots, X_n) - \overline{f}$$

and

$$V = G_j(X_1, \cdots, X_n),$$

5.6 becomes

$$U + Vt > 0. \tag{5.7}$$

We want to discard the part that makes 5.7 true. In other words, we want to retain the part that makes

$$U + Vt \leq 0 \tag{5.8}$$

true.

Let the solution set of 5.8 be T, then according to Hansen (1992), with $U = [a, b], V = [c, d]$, T is given by

$$T = \begin{cases} [-a/d, \infty] & \text{if } a \leq 0 \text{ and } d < 0 \\ [-a/c, \infty] & \text{if } a > 0, c < 0, \text{ and } d \leq 0 \\ [-\infty, \infty] & \text{if } a \leq 0 \text{ and } c \leq 0 \leq d \\ [-\infty, -a/d] \cup [-a/c, \infty] & \text{if } a > 0 \text{ and } c < 0 < d \\ [-\infty, -a/c] & \text{if } a \leq 0 \text{ and } c > 0 \\ [-\infty, -a/d] & \text{if } a > 0, c \geq 0, \text{ and } d > 0 \\ \text{empty set} & \text{if } a > 0 \text{ and } c = d = 0 \end{cases} \tag{5.9}$$

Then X_j should be replaced by $X_j \cap (T + x_{j0})$. If this set is empty, we delete all of X. When computing for dimension $j+1$, we use the improved intervals $X_1, \cdots, X_j$. This process is repeated for all $j = 1, 2, \cdots, n$.

Note from 5.9 that T can be a union of two intervals, hence the updated interval $X_j \cap (T + x_{j0})$ can be a union of two intervals as well. Computationally this is inconvenient. So in case this occurs, we simply retain the whole X_j.

5.3 Applications

In this section we give some examples to demonstrate the performance of interval optimization and discuss particular concerns with applications of interval optimization in various statistical areas.

5.3.1 Nonlinear Regression

Different models arise in different applications of statistics, resulting in a wide variety of regression problems. They provide a rich set of problems for assessing the properties of our interval optimization algorithm. We provide several examples in this area.

Example 1: Let's first look at the example we discussed in Section 5.1. We had a problem distinguishing local optimum from global optimum when using traditional methods. The data set has 2 variables and 4 observations, as in Table 5.1, and the objective function is

$$\sum_{i=1}^{4} \left(y_i - \alpha e^{-\beta x_i}\right)^2.$$

It is differentiable and the first order partial derivatives are

$$\begin{aligned}\frac{\partial}{\partial\alpha} &= -2\sum_{i=1}^{4}\left(y_i - \alpha e^{-\beta x_i}\right)e^{-\beta x_i},\\ \frac{\partial}{\partial\beta} &= 2\sum_{i=1}^{4}\left(y_i - \alpha e^{-\beta x_i}\right)\alpha x_i e^{-\beta x_i}.\end{aligned}$$

We incorporate these into the program and set the starting region to be $[-0.2, 0.3]$ $\times[-1.2, 1.3]$. When we set the tolerance for the global minimum function value inclusion to be 10^{-6}, the output region, after grouping, is

$$\hat{\alpha} \in [0.087190690636634816, 0.087190742790698994]$$
$$\hat{\beta} \in [0.619906011223793070, 0.619906271994113970],$$

and the global optimal error sum of squares value is in

$$[1.6901510372797732, 1.6901515515132814].$$

Combining what we know in Section 5.1, the result bounds the global minimum point. The existence of the local optimal point does not bother the interval optimization routine. If we set the tolerance value smaller, the length of the interval inclusions will be even shorter.

Example 2: This example is problem 10.15 in Kennedy and Gentle (1980). The data, containing more observations than in Example 1, are shown in Table 5.3.

Here we have 3 parameters to estimate, and the model is

$$y_i = \theta_1 + \frac{u_i}{\theta_2 v_i + \theta_3 w_i} + \epsilon_i$$

We know the least squares estimates of the parameters are

Table 5.3: Data set for optimization example 2

i	y_i	u_i	v_i	w_i
1	0.14	1	15	1
2	0.18	2	14	2
3	0.22	3	13	3
4	0.25	4	12	4
5	0.29	5	11	5
6	0.32	6	10	6
7	0.35	7	9	7
8	0.39	8	8	8
9	0.37	9	7	7
10	0.58	10	6	6
11	0.73	11	5	5
12	0.96	12	4	4
13	1.34	13	3	3
14	2.10	14	2	2
15	4.39	15	1	1

$$\hat{\underline{\theta}} = (0.08241, 1.1330, 2.3437)'$$

and the global minimum error sum of squares is 8.214877×10^{-3}. Calling interval optimization routines with initial region $[-10, 10] \times [0, 10] \times [0.01, 10]$ gives us the area that contains the theoretically true least squares estimate:

$$\begin{aligned} & [0.082404613494873047, 0.082415342330932617] \\ \times\ & [1.13289475440979, 1.133122444152832] \\ \times\ & [2.3436129951477054, 2.3438335644816766] \end{aligned}$$

and the global minimum error sum of squares is guaranteed to be in the interval

$$[0.0081971529628926901,\ 0.0082328356225581725].$$

Of course if the tolerance is set to be smaller, results will be better. The initial region is chosen so that θ_2 and θ_3 can not be zero at the same time. So we set the

initial region for θ_3 to be [0.01, 10] instead of [0, 10], otherwise interval analysis will produce an interval containing 0 as denominator.

Example 3: This is another example generated to demonstrate the ability of interval optimization to overcome local optima.

Consider the function

$$y = x^2 + \frac{1}{x^2},$$

which has the plot in Figure 5.6.

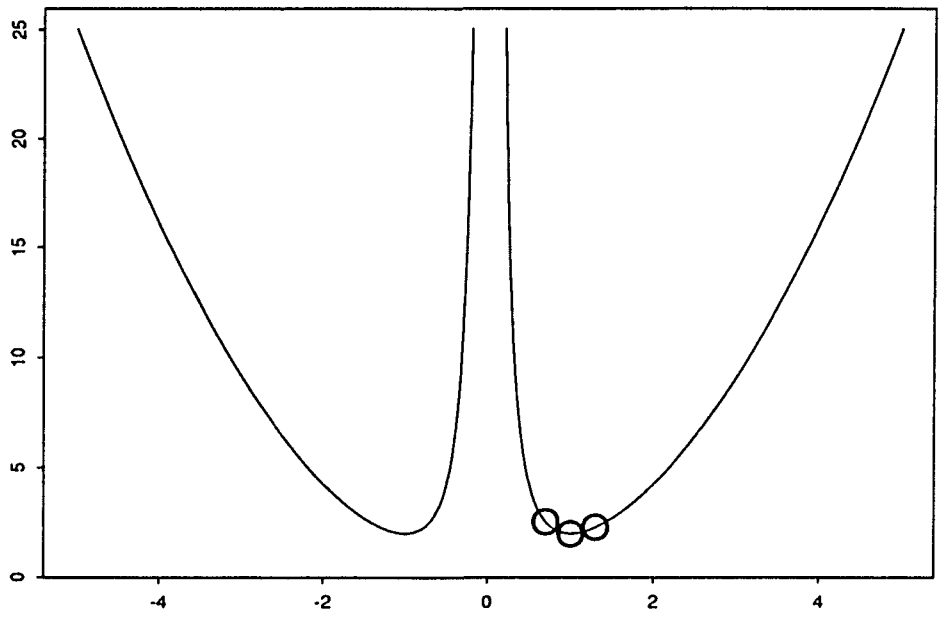

Figure 5.6: Plot of the function $y = x^2 + \frac{1}{x^2}$.

It is symmetric about $x = 0$. Suppose we sample at the circled places. They are all on the right branch of the plot. Now consider the following model to fit these data:

$$y_i = (x_i - \theta)^2 + \frac{1}{(x_i - \theta)^2} + \epsilon_i. \tag{5.10}$$

Here θ is a location parameter. The plot of 5.10 can actually be generated by shifting the plot in Figure 5.6 along the x axis. Since the data are sampled according to $\theta = 0$, the best fit to the data should be $\theta = 0$, hence it is the global minimizer to the objective function

$$\sum_{i=1}^{n} \left[y_i - (x_i - \theta)^2 - \frac{1}{(x_i - \theta)^2} \right]^2.$$

Now let's shift the plot in Figure 5.6 to the right. Because the two branches are very similar, at about $\theta = 2$ the left branch will be moving in to almost the position of the original right branch. This is shown in Figure 5.7.

Hence at around $\theta = 2$ the model will fit the data fairly well. Notice that when the plot is moved away from the position in Figure 5.7, either to the right or to the left, the model will be less perfectly fitted, and the error sum of squares will increase. Hence at around $\theta = 2$ there exists another local optimum. But the left branch is not exactly the same as the right branch, the fit will not be as perfect as the $\theta = 0$ case. So it is not a global optimum.

From the above analysis, we know that model 5.10, for the data indicated in Figure 5.6, has a global fit at $\theta = 0$ and a local fit at around $\theta = 2$. We ran both SAS *proc nlin* and interval optimization on this problem. The data set is in Table 5.4.

The behavior of SAS *proc nlin* on this problem is strange. When starting at $\theta = 0.5$, it will converge to $\theta = 8 \times 10^{-6}$. When starting at $\theta = 1.5$, it will converge

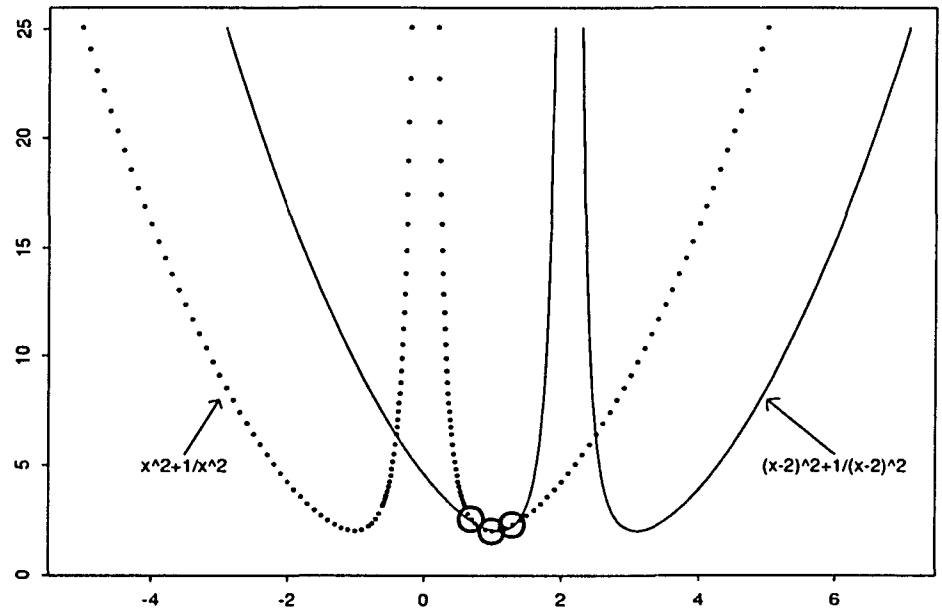

Figure 5.7: The plot of $y = (x-\theta)^2 + \dfrac{1}{(x-\theta)^2}$ when $\theta = 2$.

to $\theta = 2.04$. And when starting at $\theta = 1.1$ or 0.9, it will just stay there with very large error sum of squares. This is because at around $\theta = 1$, the function value is ∞ at $x = 1$. And all the data points are grouped around $x = 1$, making the error sum of squares huge and the behavior of SAS *proc nlin* abnormal.

Now let's look at interval optimization. We use the starting region $[-100, 100]$ for θ, and it produces only one global minimum point

Table 5.4: Data set for optimization example 3.

x_i	y_i
0.8	2.2025
1.0	2.0
1.2	2.1344

$$\hat{\theta} \in [0.000008009374141693 1152, 0.000008079223334789 2761],$$

and the error sum of squares at this point is in interval

$$[0.000000001515280606 4282, 0.000000001546691320 5340].$$

Hence it overcomes the presence of the local optimum very nicely.

Example 4: This example demonstrates interval optimization's ability to find multiple global optimal points at the same time.

Consider the function

$$z = x^2y^2 + \frac{1}{x^2y^2}. \tag{5.11}$$

When we plot 5.11, on every of the 4 quadrants above the plane $z = 2$ it will be valley-shaped. The bottoms of the valleys are shown in Figure 5.8. Because 5.11 is an even function on both x and y, the four valleys should be identical in shape.

Suppose we sample on the valley in the first quadrant. We obtain the data set as in Table 5.5.

Consider fitting the data with model

$$z_i = (\alpha x_i + \beta y_i)^2(-\beta x_i + \alpha y_i)^2 + \frac{1}{(\alpha x_i + \beta y_i)^2(-\beta x_i + \alpha y_i)^2} + \epsilon_i. \tag{5.12}$$

This is obtained from 5.11 by changing

$$\begin{bmatrix} x \\ y \end{bmatrix} \quad \text{to} \quad \begin{bmatrix} \alpha & \beta \\ -\beta & \alpha \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

In case $\alpha^2 + \beta^2 = 1$, we are actually doing a rotation. Note that $\alpha = 1$ and $\beta = 0$ give us just 5.11. Hence, since the data is sampled from the function curve of 5.11, $\alpha = 1, \beta = 0$ should be a perfect fit and hence a global optimum.

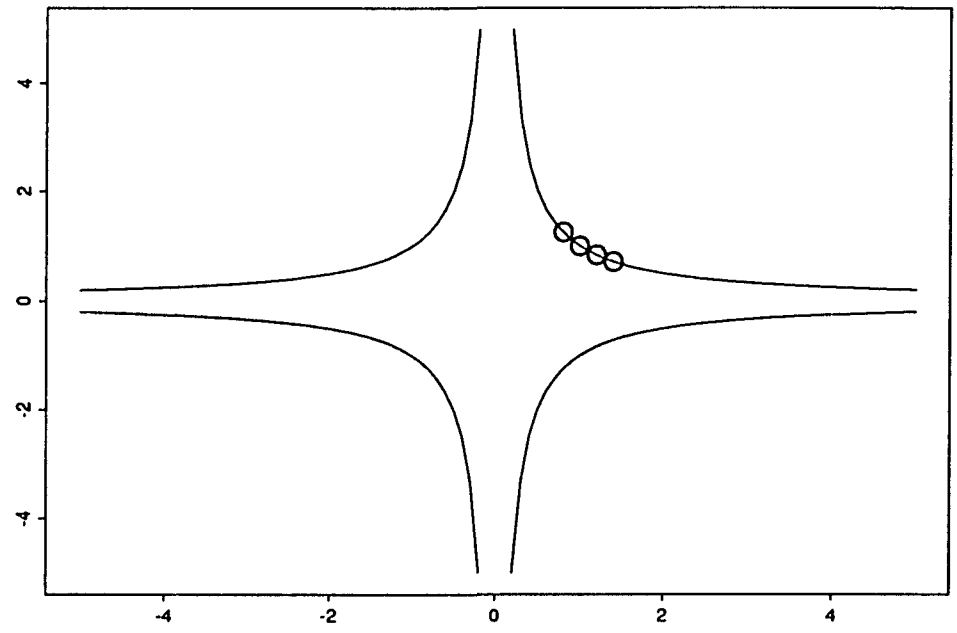

Figure 5.8: The bottom of the valley of the function $z = x^2y^2 + \frac{1}{x^2y^2}$.

Note also that the four valleys for 5.11 are identical in shape. When we fit the model 5.12, or in other words, rotate the four identical valleys, perfect fits will occur. From Figure 5.8 we know that rotating $\pi/2, \pi$ or $3\pi/2$ will give us exactly the same contour plots as 5.11. Hence the 3 sets of parameters shown in Table 5.6 can fit the data just as well as $\alpha = 1, \beta = 0$, and are also global optimum points.

Table 5.5: Data set for optimization example 4.

z_i	x_i	y_i
2.2025	1.0	0.8
2.0	1.0	1.0
2.2025	0.8	1.0
2.18	0.9	0.9

Table 5.6: Other global optimal points for example 4.

Rotation Angle	α	β
$\pi/2$	0	1
π	-1	0
$3\pi/2$	0	-1

So we know that the model 5.12 for the data in Table 5.5 has four global optimal fits.

Using SAS *proc nlin*, we get only one global point at a time. The results are summarized in Table 5.7.

Table 5.7: SAS results for optimization example 4.

Starting Point	$\hat{\alpha}$	$\hat{\beta}$
$(0.8, 0.2)$	1.000022795	0.000000001
$(-0.8, 0.2)$	-1.000022795	0.000000001
$(0.2, 0.8)$	0.000000001	1.000022795
$(0.2, -0.8)$	0.000000001	-1.000022795

But on MP2, using interval optimization we obtain all four global optima at the same time.

$$\hat{\alpha}_1 \in [0.99999973080806159000, 1.00000000000000000000]$$

$$\hat{\beta}_1 \in [-0.00000667572021484375, 0.00000667572021484375]$$

$$SSE_1 \in [0.000000039592241891 1422, 0.000000111622167753 0891]$$

$$\hat{\alpha}_2 \in [-0.00000667572021484375, 0.00000667572021484375]$$

$$\hat{\beta}_2 \in [0.99999973080806159000, 1.00000000000000000000]$$

$$SSE_2 \in [0.000000039592241891 1422, 0.000000111622167753 0891]$$

$$\hat{\alpha}_3 \in [-1.00000000000000000000, -0.99999952316284180000]$$

$$\begin{aligned}
\hat{\beta}_3 &\in [-0.00000667572021484375, 0.00002288818359375000] \\
SSE_3 &\in [0.000000016317977266020 0, 0.000000206837089420 1559] \\
\hat{\alpha}_4 &\in [-0.00000667572021484375, 0.00000667572021484375] \\
\hat{\beta}_4 &\in [-1.00000000000000000000, -0.99999973080806159000] \\
SSE_4 &\in [0.000000039592241891 1422, 0.000000111622167753 0891]
\end{aligned}$$

5.3.2 Optimal Design

Suppose a random variable y is measured at n design points $\underline{x}_1, \underline{x}_2, \cdots, \underline{x}_n$ where each $\underline{x}_i$ consists of m independent variables. Suppose further that the relationship between y_i and $\underline{x}_i$ can be described by the following linear model

$$y_i = \underline{f}'(\underline{x}_i)\underline{\beta} + \epsilon_i \tag{5.13}$$

where $\underline{\beta}$ is a $p \times 1$ vector of unknown parameters, and $\underline{f}(\underline{x})$ is a $p \times 1$ vector function on $\underline{x}$. In matrix notation, 5.13 can be written as

$$\underline{y} = X\underline{\beta} + \underline{\epsilon}$$

where X is the design matrix:

$$\begin{bmatrix} \underline{f}'(\underline{x}_1) \\ \vdots \\ \underline{f}'(\underline{x}_n) \end{bmatrix} \tag{5.14}$$

and

$$E(\underline{\epsilon}) = \underline{0}, \qquad Var(\underline{\epsilon}) = \sigma^2 I.$$

If X is of full column rank, then from linear model theory, we know the least squares estimate of $\underline{\beta}$ and its variance-covariance matrix are

$$\underline{\hat{\beta}} = (X'X)^{-1}X'\underline{y},$$
$$Var(\underline{\hat{\beta}}) = (X'X)^{-1}\sigma^2.$$

The goal here is to choose the n design points to make the estimate of $\underline{\beta}$ as precise as possible. Thus we want to make $Var(\underline{\hat{\beta}})$ as small as possible. But there is no unique ranking of matrices. Hence different criteria arise as mentioned at the beginning of this chapter. A more complete description can be found in Pukelsheim (1993). In this section we will focus on the D-optimal designs, i.e., minimizing the variance of estimation in the sense of minimizing the generalized variance of $\underline{\hat{\beta}}$, $\left|(X'X)^{-1}\right|$, or equivalently, maximizing $|X'X|$. D-optimality is used most frequently by experiment designers because it has a nice property of invariance to scaling. See Pukelsheim (1993).

Box and Lucas (1959) extended the concept of D-optimality to the cases of nonlinear models. Now the model becomes

$$y_i = f(\underline{x}_i, \underline{\beta}) + \epsilon_i$$

and the D-optimal design maximizes $|X'X|$ where the ijth element of X is

$$X_{ij} = \frac{\partial}{\partial \beta_j} f(\underline{x}_i, \underline{\beta})\Big|_{\underline{\beta}=\underline{\beta}_0}, \quad i = 1, \cdots, n, \quad j = 1, \cdots, p. \tag{5.15}$$

Here $\underline{\beta}_0$ is some value for the unknown parameter $\underline{\beta}$. For nonlinear models the D-optimal design depends upon the actual values of the p-parameters. If we are to suppose that effective design is possible at all, we must also assume therefore that something is known about the values of the parameters in advance. In practice this is usually not of much trouble.

Now our objective function in maximization becomes $|X'X|$. One may think that with initial regions of the $\underline{x}_i$'s we can just use interval analysis to get inclusions of the elements of X according to 5.14 or 5.15, then use, say, Gaussian elimination, to get $|X'X|$.

But this very often will get us into trouble. The reason is the data dependency problem. In using interval analysis to evaluate an expression, if some interval variable appears more than once, the result might not be as tight as we want. For example, if we evaluate $(X - Y)/(X + Y)$ using $X = [2,3], Y = [1,2]$, we get $[0,2/3]$; but the exact range of it is $[0,1/2]$, which can be obtained by evaluating $1 - 2/(1 + X/Y)$. Hence we want to find the best way to perform our computation in order to minimize the data dependency problem.

Now look at $|X'X|$. When forming $X'X$, each element is used p times. There is heavy data dependency. After this, the Gaussian elimination on $X'X$ again involves very much data dependency. Hence we want to develop more suitable algorithm to deal with the computation of $|X'X|$. Since the formation of $X'X$ has data dependency, we consider a method that will compute $|X'X|$ without forming $X'X$ first. In other words, we look for a method to obtain $|X'X|$ directly from X.

Suppose there is an orthogonal matrix P such that PX is an upper triangular matrix Q. Now

$$|Q'Q| = |(PX)'PX| = |X'P'PX| = |X'X|.$$

Suppose

$$Q = \begin{bmatrix} Q_1 \\ 0 \end{bmatrix}$$

where Q_1 is an upper triangular square matrix. Then

$$|Q'Q| = [Q_1'0] \begin{bmatrix} Q_1 \\ 0 \end{bmatrix} = |Q_1'Q_1| = |Q_1|^2$$

Denote the diagonal elements of Q by $q_{11}, q_{22}, \cdots, q_{pp}$, then

$$|Q_1| = \prod_{i=1}^{p} q_{ii}$$

and

$$|X'X| = |Q'Q| = |Q_1|^2 = \left(\prod_{i=1}^{p} q_{ii}\right)^2.$$

Therefore, we can first use an orthogonal transformation to find P and Q, and then just multiply the diagonal elements of Q. One of the orthogonal transformations is the Householder transformation. We implemented it in a subroutine.

Other algorithms we tried are the Cholesky decomposition (Kennedy and Gentle, 1980), direct Gaussian elimination and modified Gaussian elimination, which we call Hansen's method because it was mentioned in Hansen and Smith (1967). It proceeds by forming $X'X$ first, then form a scalar matrix with elements being the center points of each interval element and perform LU factorization on it. Then multiply $X'X$ by L^{-1} to get $L^{-1}X'X$. Since the diagonal elements of L are all 1, the multiplication will not change the determinant value. But it is claimed that after the multiplication the elements below the main diagonal will have small intervals. Hence it will reduce the seriousness of data dependency. "Small" is in the sense of magnitudes of endpoints.

In addition to these four algorithms to compute $|X'X|$, for low dimensional X we also developed special algorithms. They compute $|X'X|$ directly by expanding $|X'X|$ first. This involves less computation hence we expect less severe data dependency problem.

We have these routines to perform the computation for $|X'X|$. Experience shows that for high-dimensional $X'X$, data dependency still can not be avoided. Also, if we choose to form $X'X$ first, the resulting symmetric interval matrix will contain many unsymmetric scalar matrices. For example, the ijth and jith elements of $X'X$ are the same interval, but we can choose two different values from this interval for the ijth and jith elements of a scalar matrix that is contained in $X'X$. These unsymmetric matrices are making the subsequent computation results wide when symmetric matrix operations are performed on them (like Householder transformation).

For the derivative information that we have to provide to the interval optimization routine, we have to compute

$$\frac{\partial}{\partial X_{st}} |X'X|, \quad s = 1, 2, \cdots, n, \quad t = 1, 2, \cdots, p$$

for each element X_{st} of X, then the partial derivative of $|X'X|$ with respect to the original m variables can be obtained by chain rules. Note that the partial derivatives of X_{st} with respect to the original variables can be easily derived.

We first compute derivatives of $X'X$ with respect to the elements of $X'X$. For any element $(X'X)_{ij}$ of $X'X$, according to Bates (1983),

$$\frac{\partial}{\partial (X'X)_{ij}} |X'X| = |X'X| (X'X)^{-1}_{ij},$$

where $(X'X)^{-1}_{ij}$ is the ijth element of $(X'X)^{-1}$. Then

$$\begin{aligned}
\frac{\partial}{\partial X_{st}} |X'X| &= \sum_{i=1}^{p} \sum_{j=1}^{p} \left[\frac{\partial}{\partial (X'X)_{ij}} |X'X| \cdot \frac{\partial}{\partial X_{st}} (X'X)_{ij} \right] \\
&= |X'X| \sum_{i=1}^{p} \sum_{j=1}^{p} \left[(X'X)^{-1}_{ij} \frac{\partial}{\partial X_{st}} (X'X)_{ij} \right] \\
&= |X'X| \sum_{j=1}^{p} \left[(X'X)^{-1}_{tj} \frac{\partial}{\partial X_{st}} (X'X)_{tj} \right]
\end{aligned}$$

$$
\begin{aligned}
&+|X'X|\sum_{i=1}^{p}\left[(X'X)^{-1}_{it}\frac{\partial}{\partial X_{st}}(X'X)_{it}\right] \\
&-|X'X|\left[(X'X)^{-1}_{tt}\frac{\partial}{\partial X_{st}}(X'X)_{tt}\right] \\
=\ &|X'X|\sum_{j=1}^{p}\left[(X'X)^{-1}_{tj}\frac{\partial}{\partial X_{st}}(X'X)_{tj}\right] \\
&+|X'X|\sum_{i=1}^{p}\left[(X'X)^{-1}_{it}\frac{\partial}{\partial X_{st}}(X'X)_{it}\right] \\
&-|X'X|\left[(X'X)^{-1}_{tt}\cdot 2X_{st}\right] \\
=\ &|X'X|\sum_{j=1}^{p}\left[(X'X)^{-1}_{tj}X_{sj}\right]+|X'X|\sum_{i=1}^{p}\left[(X'X)^{-1}_{it}X_{si}\right] \\
=\ &2|X'X|\sum_{i=1}^{p}\left[(X'X)^{-1}_{it}X_{si}\right] \\
=\ &2\sum_{i=1}^{p}\left[(X'X)^{*}_{it}X_{si}\right]. \qquad (5.16)
\end{aligned}
$$

Here $(X'X)^*$ denotes the cofactor matrix of $X'X$.

Let $\underline{X}'_i$ be the ith row of X. Then from 5.16 we have

$$\frac{\partial}{\partial \underline{X}'_i}|X'X| = 2(X'X)^*\underline{X}'_i.$$

We will use this relation to compute the partial derivative information. Note also that in the high dimensional cases the computation of the cofactor matrix involves heavy data dependency. We do not expect tight interval inclusions for the derivatives in those cases.

For low dimensional $X'X$ things are going well, providing us with good test examples for interval optimization.

Example 5: This example comes from Haines (1987). We have a polynomial regression model of degree 2:

$$y_i = \underline{f}'(x_i)\underline{\beta} + \epsilon_i, i = 1, 2, \cdots, n.$$

where $\underline{f}'(x_i) = (1\ x_i\ x_i^2)$ and $x_i \in [-1, 1]$.

Think about a 3-point design. The X matrix becomes

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{bmatrix}$$

and $\underline{\beta}$ is 3-dimensional:

$$\underline{\beta} = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix}.$$

For the D-optimal design $\underline{x} = (x_1, x_2, x_3)$, if we permute the 3 points the results should all be D-optimal designs because there is no ordering among the 3 points. Therefore we can expect 6 global optimal points. Using interval optimization with initial regions $[-1, 1] \times [-1, 1] \times [-1, 1]$, we get them at the same time.

$$\begin{aligned}
x_1 &\in [-0.000732421875, 0.000976562500] \\
x_2 &\in [-1.000000000000, -0.999511718750] \\
x_3 &\in [0.999511718750, 1.000000000000] \\
|X'X|_1 &\in [3.9648558918703984, 4.0234332166332392] \\
x_1 &\in [0.999023437500, 1.000000000000] \\
x_2 &\in [-0.001464843750, 0.001953125000] \\
x_3 &\in [-1.000000000000, -0.999023437500] \\
|X'X|_2 &\in [3.9297359974545101, 4.0468578993521493]
\end{aligned}$$

$$
\begin{aligned}
x_1 &\in [-1.000000000000, -0.999511718750] \\
x_2 &\in [-0.000488281250, 0.000976562500] \\
x_3 &\in [0.999511718750, 1.000000000000] \\
|X'X|_3 &\in [3.9648558918717627, 4.0234325013764796] \\
x_1 &\in [0.999511718750, 1.000000000000] \\
x_2 &\in [-1.000000000000, -0.999511718750] \\
x_3 &\in [-0.000976562500, 0.000976562500] \\
|X'X|_4 &\in [3.9648558918690342, 4.0234339318899988] \\
x_1 &\in [-1.000000000000, -0.999511718750] \\
x_2 &\in [0.999511718750, 1.000000000000] \\
x_3 &\in [-0.000488281250, 0.000976562500] \\
|X'X|_5 &\in [3.9648558918717627, 4.0234325013764796] \\
x_1 &\in [-0.000976562500, 0.000976562500] \\
x_2 &\in [0.999511718750, 1.000000000000] \\
x_3 &\in [-1.000000000000, -0.999511718750] \\
|X'X|_6 &\in [3.9648558918690342, 4.0234339318899988]
\end{aligned}
$$

Hence the D-optimal design requires 3 points : -1, 0 and 1.

Example 6: This example is from Box and Lucas (1959). It demonstrates optimal design in nonlinear model cases. Suppose a consecutive chemical reaction is under study in which a substance A decomposes to form substance B which then in turn decomposes to form substance C. With the assumptions that the reactions are first

order and irreversible, it may be shown that, after time ξ has elapsed, the yield η of intermediate product B is given by

$$\eta = \frac{\theta_1}{\theta_1 - \theta_2}\left(e^{-\theta_2\xi} - e^{-\theta_1\xi}\right) \tag{5.17}$$

where θ_1 and θ_2 are constants measuring the specific rates of the first and second decompositions, respectively.

The problem is to choose a set of times $\xi_i, i = 1, 2, \cdots, n$, at which to observe the yield so that from those observations θ_1 and θ_2 can be estimated as accurately as possible. According to the formation of the X matrix in a nonlinear model case, we have to find the derivatives of 5.17 with respect to θ_1 and θ_2:

$$\Phi_1(\xi) = \frac{\partial}{\partial\theta_1}\eta = \frac{\left({\theta_1}^2\xi + \theta_2 - \theta_1\theta_2\xi\right)e^{-\theta_1\xi} - \theta_2 e^{-\theta_2\xi}}{(\theta_1 - \theta_2)^2},$$
$$\Phi_2(\xi) = \frac{\partial}{\partial\theta_2}\eta = \frac{-\theta_1 e^{-\theta_1\xi} + \left(\theta_1 - {\theta_1}^2\xi + \theta_1\theta_2\xi\right)e^{-\theta_2\xi}}{(\theta_1 - \theta_2)^2}.$$

In the case of $n = 2$, we plug in our preliminary guesses $\theta_1^* = 0.7, \theta_2^* = 0.2$, and obtain the X matrix

$$X = \begin{bmatrix} \Phi_1(\xi_1) & \Phi_2(\xi_1) \\ \Phi_1(\xi_2) & \Phi_2(\xi_2) \end{bmatrix}$$
$$= \begin{bmatrix} (0.8 + 1.4\xi_1)e^{-0.7\xi_1} - 0.8e^{-0.2\xi_1} & (2.8 - 1.4\xi_1)e^{-0.2\xi_1} - 2.8e^{-0.7\xi_1} \\ (0.8 + 1.4\xi_2)e^{-0.7\xi_2} - 0.8e^{-0.2\xi_2} & (2.8 - 1.4\xi_2)e^{-0.2\xi_2} - 2.8e^{-0.7\xi_2} \end{bmatrix}.$$

We then proceed with interval optimization. In theory, as discussed in Box and Lucas (1959), there is a global maximum point at $\xi_1 = 1.23, \xi_2 = 6.86$, as well as a local maximum point at $\xi_1 = 7.5, \xi_2 = 15.0$. This may cause some difficulty to traditional numerical methods. But interval optimization gets the following results:

$$\xi_1 = [6.85768890380859380, 6.85768914222717290]$$

$$\begin{aligned}
\xi_2 &= [1.22947135567665100, 1.22947144508361820] \\
|X'X|_1 &= [0.65677365189005421, 0.65677420980082812] \\
\xi_1 &= [1.2294713705778122, 1.229471430182457] \\
\xi_2 &= [6.8576889634132385, 6.8576891422271729] \\
|X'X|_2 &= [0.65677373856361776, 0.65677412312722605]
\end{aligned}$$

It gets two global maximum points because we can exchange ξ_1 and ξ_2. The local maximum is nicely thrown away.

5.3.3 Moving Average Model

Example 7: Let's consider the first order moving average model MA(1) of time series defined by

$$y_t = u_t + \alpha u_{t-1}$$

where $|\alpha| < 1$ and the u_t are iid $N(0, \sigma^2)$. Reparameterize according to

$$\sigma_0 = \sigma^2 \left(1 + \alpha^2\right)$$

and

$$\rho = \frac{\alpha}{1 + \alpha^2},$$

then in terms of σ_0 and ρ, the likelihood function of a vector $\underline{y} = (y_1, \cdots, y_T)'$ of observations is

$$L(\sigma_0, \rho) = (2\pi\sigma_0)^{-T/2} |R|^{-1/2} exp\left(-\underline{y}' R^{-1} \underline{y}/(2\sigma_0)\right), \tag{5.18}$$

where R is the $T \times T$ matrix of autocorrelations. It has 1's along the main diagonal, ρ's along the two diagonals adjacent to the main diagonal, and 0's elsewhere. The goal is to maximize 5.18 with respect to σ_0 and ρ.

We used SAS function *rannor* to generate iid $N(0,1)$ random variables $u_t, t = 1, 2, \cdots, 16$, as shown in Table 5.8.

Table 5.8: Data for the iid $N(0,1)$ random variables for example 7.

t	u_t
1	0.56707
2	0.98383
3	0.62149
4	1.31212
5	−0.18328
6	1.32202
7	−0.15609
8	0.23049
9	0.13378
10	0.36692
11	−0.43875
12	1.01097
13	−0.32892
14	−0.71748
15	−1.48650
16	0.41967

Then we used a C program with $\alpha = 0.3$ to generate the y_t's, $t = 1, 2, \cdots, 15$. The data is shown in Table 5.9.

We want to use these data in 5.18 to perform our optimization. The parameter values that the data is based on are $\sigma^2 = 1$ and $\alpha = 0.3$. Hence $\sigma_0 = 1.09$ and $\rho = 0.27523$. But because the data is generated as a random variable, we will not be surprised to get MLEs other than the above mentioned values.

Table 5.9: Data for the random vector $\underline{y}$ for example 7.

t	y_t
1	1.153951
2	0.916639
3	1.498567
4	0.210356
5	1.267036
6	0.240516
7	0.183663
8	0.202927
9	0.407054
10	−0.328674
11	0.879345
12	−0.025629
13	−0.816156
14	−1.701744
15	−0.026280

Because R in 5.18 has a special tridiagonal form, the computation of $|R|$ has an explicit form according to Anderson (1971):

$$|R| = a^{-1}\left(\frac{1}{2}\right)^{T+1}\left[(1+a)^{T+1} - (1-a)^{T+1}\right]$$

where $a = (1-4\rho^2)^{1/2}$. Also in the program since we don't have a general explicit form for R^{-1}, which involves ρ, we can not get the derivative for ρ. Hence we used only the derivative information for σ_0.

The interval optimization result, when using starting values $\sigma_0 \in [0.1, 50]$ and $\rho \in [-0.49, 0.49]$, is shown below :

$$\hat{\sigma_0} \in [0.51037812232971191, 0.51182361841201773]$$

$$\hat{\rho} \in [0.18019851684570312, 0.18749588012695315]$$

$$L(\hat{\sigma_0}, \hat{\rho}) \in [0.00037604020886129107, 0.00048138582659166553]$$

The SAS *proc arima*, using maximum likelihood method, gives us

$$\tilde{\sigma_0} = 0.5476$$

$$\tilde{\rho} = 0.18396$$

$\tilde{\rho}$ is contained in the interval for $\hat{\rho}$, but $\tilde{\sigma_0}$ is outside the interval for the global optimizer. The reason could be either that $\tilde{\sigma_0}$ is just a local optimum, or that SAS computation is too inaccurate. The use of interval optimization takes into account both of them and generates a guaranteed range for the global optimizer.

CHAPTER 6. CONCLUSIONS

In this dissertation we introduced interval analysis and its applications in statistics. Interval analysis is trying to use intervals, instead of scalars, as computing elements. At every computing step, the theoretically true real number result is guaranteed to be contained in the computed interval, despite the inaccuracy of the computer number system and the uncertainty of rounding and approximation errors. This idea, with the support from interval arithmetic, provides us with a way to control the accuracy of the computation. The essential mechanism behind it is the capability of interval analysis to compute the range of a function over a certain region. We showed two major applications of it in this dissertation: to obtain guaranteed results for multivariate probabilities and guaranteed global optimizers in optimization problems.

In multivariate Normal and multivariate t probability computation problems, we first expand the integrand to its Taylor expansion. Then we analyze both the rule part and the remainder part by means of interval analysis to get interval inclusions. With the help of automatic differentiation and the parallel computer MasPar, we were able to develop a systematic algorithm for the general n-variate Normal and t distribution on finite regions. In tolerable running time we obtain guaranteed interval result to contain the true integration value we want to compute. The length of the interval we

obtained ranges from 10^{-6} in 10-dimensional cases to 10^{-15} in 6-dimensional cases for the integration region being $[-0.5, 0.5]$ on each dimension. For lower dimensional integration the results will be even better. This part of the research basically answers the demand of highly reliable algorithms for this kind of computation. It can be used in two cases: in some crucial situation where we want high accuracy results, and in general as a comparing standard for testing the accuracy of other software packages.

In global optimization, currently there is no software that can guarantee to find the global optimum. People just take whatever they obtain as a good enough result. The effectiveness of these results depends heavily on the initial values, as shown by the examples in Chapter 5. Interval optimization, on the other hand, considers every point in the whole feasible region instead of just isolated points at a time. This way we will not lose track of any point. Specifically, we split the whole feasible region into small pieces and use the capability of interval analysis to compute inclusions of the range of the objective function over these subregions. After these inclusions are obtained, we use three criteria to tell whether or not a region can not contain the global optimizer and hence throw away that region, or part of it. For the remaining smaller region we do the similar thing, until the regions are sufficiently small in size. We took advantage of the MasPar machine and made the algorithm even more efficient. From the examples provided in Chapter 5, we can see that the performance of interval optimization is much better than traditional algorithms. It will not be bothered by local optima. And if there exist more than one global optimum, it is capable of locating them all at the same time. Interval optimization can be used in various statistical areas, like nonlinear regression, experimental design, etc.

Overall, interval analysis in statistical computing is proved to be a powerful

tool. With modern high performance computers it becomes even more feasible and practical to use. We studied several applications of it, and believe it can be applied in other areas as well.

BIBLIOGRAPHY

Aberth, Oliver (1988), *Precise Numerical Analysis*, Dubuque, Iowa: Wm. C. Brown Publishers.

Anderson, T. W. (1971), *The Statistical Analysis of Time Series*, New York: Wiley.

Bard, Yonathan (1974), *Nonlinear Parameter Estimation*, New York: Academic Press.

Bates, Douglas (1983), The Derivative of $|X'X|$ and Its Uses, *Technometrics*, 25, 373-376.

Bohachevsky, Ihor O., Johnson, Mark E. and Stein, Myron L. (1986), Generalized Simulated Annealing for Function Optimization, *Technometrics*, 28, 209-217.

Box, G. E. P. and Lucas, H. L. (1959), Design of Experiments in Non-Linear Situations, *Biometrika*, 46, 77-90.

Clemmesen, Michael (1984), Interval Arithmetic Implementations Using Floating Point Arithmetic, *ACM SIGNUM Newsletter*, 19, No. 4, 2-8.

Cook, R. Dennis and Nachtsheim, Christopher J. (1980), A Comparison of Algorithms for Constructing Exact D-Optimal Designs, *Technometrics*, 22, 315-324.

Cook, R. Dennis and Nachtsheim, Christopher J. (1989), Computer-Aided Blocking of Factorial and Response-Surface Designs, *Technometrics*, 31, 339-346.

Corliss, George F. and Rall, L. B. (1987), Adaptive, Self-Validating Numerical Quadrature, *SIAM J. Sci. Stat. Comput.*, 8, 831-847.

Davis, Philip J. and Rabinowitz, Philip (1975), *Methods of Numerical Integration*, New York: Academic Press.

Dodge, Yadolah, Fedorov, Valeri V. and Wynn, Henry P. (1988), Optimal Design of Experiments: An Overview, *Optimal Design and Analysis of Experiments*, ed. Yadolah Dodge, Valeri V. Fedorov and Henry P. Wynn, Amsterdam: North-Holland.

Drezner, Zvi (1992), Computation of the Multivariate Normal Integral, *ACM Trans. on Math. Soft.*, 18, 470-480.

Everitt, B. S. (1987), *Introduction to Optimization Methods and Their Application in Statistics*, London: Chapman and Hall.

Faddeev, D. K. and Faddeeva, V. N. (1963), *Computational Methods of Linear Algebra*, trans. Robert C. Williams, San Francisco: W. H. Freeman and Company.

Flournoy, Nancy and Tsutakawa, Robert K. ed. (1989), *Statistical Multiple Integration*, Providence, R.I.: American Mathematical Society.

Galil, Z. and Kiefer, J. (1980), Time- and Space-Saving Computer Methods, Related to Mitchell's DETMAX, for Finding D-Optimum Designs, *Technometrics*, 22, 301-313.

Genz, Alan (1992), Numerical Computation of Multivariate Normal Probabilities, *J. of Computational and Graphical Stat.*, 1, 141-149.

Gragg, W. B., Leveque, R. J. and Trangenstein, J. A. (1979), Numerically Stable Methods for Updating Regressions, *JASA*, 74, 161-168.

Griewank, Andreas and Corliss, George F. ed. (1991), *Automatic Differentiation of Algorithms: Theory, Implementation, and application*, Philadelphia: SIAM.

Haines, Linda M. (1987), The Application of the Annealing Algorithm to the Construction of Exact Optimal Designs for Linear-Regression Models, *Technometrics*, 29, 439-447.

Hammersley, J. M. and Handscomb, D. C. (1964), *Monte Carlo Methods*, New York: Wiley.

Hansen, Eldon (1992), *Global Optimization Using Interval Analysis*, New York: Marcel Dekker, Inc.

Hansen, Eldon and Smith, Roberta (1967), Interval Arithmetic in Matrix Computations, Part 2, *SIAM J. Numer. Anal.*, 4, 1-9.

Jilin University (1978), *Mathematical Analysis*, Jilin, China: People's Education Press.

Kennedy, William J. Jr. and Gentle, James E. (1980), *Statistical Computing*, New York: Marcel Dekker, Inc.

Knuth, Donald E. (1981), *The Art of Computer Programming*, 2nd edition, Vol. 2, Reading, Mass.: Addison-Wesley.

Kulisch, Ulrich W. and Miranker, Willard L. (1983), ed. *A New Approach to Scientific Computation*, New York: Academic Press.

Lu, Kaicheng (1983), *Combinatorial Mathematics: Algorithms and Analysis*, Beijing, China: Qinghua University Press.

MasPar Computer Corporation (1992a), *MasPar Programming Language User Guide*, Sunnyvale, Calif.: MasPar Computer Corporation.

MasPar Computer Corporation (1992b), *MasPar System Overview*, Sunnyvale, Calif.: MasPar Computer Corporation.

Mitchell, Toby J. (1974), An Algorithm for the Construction of D-Optimal Experimental Designs, *Technometrics*, 16, 203-210.

Moore, R. E. (1962), *Interval Arithmetic and Automatic Error Analysis in Digital Computing*, Ph.D. dissertation, Stanford University.

Moore, R. E. (1979), *Methods and Applications of Interval Analysis*, Philadelphia: SIAM.

Moore, R. E., Hansen, Eldon and Leclerc Anthony (1992), Rigorous Methods for Global Optimization, *Recent Advances in Global Optimization*, ed. Christodoulos A. Floudas and Panos M. Pardalos, Princeton, N.J.: Princeton University Press, 321-342.

Neidinger, Richard D. (1992), An Efficient Method for the Numerical Evaluation of Partial Derivatives of Arbitrary Order, *ACM Trans. on Math. Soft.*, 18, 159-173.

Piessens R., de Doncker-Kapenga E., Uberhuber C.W. and Kahaner D.K. (1983), *Quadpack*, Berlin: Springer-Verlag.

Pukelsheim, Friedrich (1993), *Optimal Design of Experiments*, New York: Wiley.

Rall, Louis B. (1981), *Automatic Differentiation: Techniques and Applications*, Berlin: Springer-Verlag.

Ratkowsky, David A. (1990), *Handbook of Nonlinear Regression Models*, New York: Marcel Dekker, Inc.

Ratschek, H. and Rokne, J. (1988), *New Computer Methods for Global Optimization*, New York: Halsted Press.

Rosen, Kenneth H. (1984), *Elementary Number Theory and Its Applications*, Reading, Mass.: Addison-Wesley.

Schervish, Mark J. (1984), Multivariate Normal Probabilities with Error Bound, *Applied Statistics*, 33, 81-94.

Scheaffer, Richard L., Mendenhall, William and Ott, Lyman (1979), *Elementary Survey Sampling*, 2nd edition, Boston: Duxbury Press.

Seber, G. A. F. and Wild, C. J. (1989), *Nonlinear Regression*, New York: Wiley.

Silvey, S. D. (1980), *Optimal Design*, London: Chapman and Hall.

Statistical Laboratory, Iowa State University (1986), *PC CARP*, Ames, Iowa.

Tong, Y. L. (1990), *The Multivariate Normal Distribution*, New York: Springer-Verlag.

Wang, Morgan and Kennedy, William J. (1990), Comparison of Algorithms for Bivariate Normal Probability over a Rectangle Based on Self-Validating Results from Interval Analysis, *J. Statist. Comput. Simul.*, 37, 13-25.

Welch, William J. (1982), Branch-and-Bound Search for Experimental Designs based on *D* Optimality and Other Criteria, *Technometrics*, 24, 41-48.

Welch, William J. (1984), Computer-Aided Design of Experiments for Response Estimation, *Technometrics*, 26, 217-224.

Youngs, Edward A. and Cramer, Elliot M. (1971), Some Results Relevant to Choice of Sum and Sum-of-Product Algorithms, *Technometrics*, 13, 657-665.

Zanakis, S. H. and Rustagi, J. S. (1982), ed. *Optimization in Statistics*, Amsterdam: North-Holland.

ACKNOWLEDGEMENTS

I wish to express my heartfelt gratitude to my major professor, Dr. William J. Kennedy, for all of his kindness, enthusiasm, invaluable guidance and continuous encouragement, without which I could not have gone thus far. Thanks also extend to professors Yasuo Amemiya, Dale Grosvenor, Kenneth J. Koehler and Mervin Marasinge for kindly serving on my committee and sharing their time and knowledge with me.

I am thankful to the Survey Section and the Department of Statistics for all the support, direct and indirect, during the course of this research, which makes my stay at Iowa State University an enjoyable one.

I am indebted to Ames Laboratory for their granting me computer time on the MasPar machines, without which this research would not be possible.

Finally, my deepest appreciation goes to my wife, Lu, for her patience, understanding and everlasting support, especially during the darkness of the early years of my graduate study.

APPENDIX A. INTERVAL ROUTINES ON MASPAR

```
/*****************************************************************************/
/*                              mytype.h (MP1)                               */
/*****************************************************************************/
typedef union { double unf;
                unsigned long uni[2];
              } singleton;
typedef struct { singleton l;
                 singleton r;
               } intvl;


/*****************************************************************************/
/*                               succ.m (MP1)                                */
/*****************************************************************************/
#include "mytype.h"
#include <stdio.h>
#define OVERFLOW p_printf("overflow\n");
plural singleton succ(x)
plural singleton x;
{ plural singleton t;
  plural int i;
  plural unsigned long tag;
  t.unf=x.unf;
  if ((t.uni[1] & 0x80000000) == 0x0)
    { tag=0x1;
      while ((t.uni[0] & tag) != 0x0 && tag != 0x0)
        { t.uni[0]&=~tag;
          tag<<=1;
        }
      if (tag != 0x0) t.uni[0]|=tag;
      else
```

```
        { tag=0x1;
          i=1;
          while ((t.uni[1] & tag) != 0x0 && i <= 31)
            { t.uni[1]&=~tag;
              tag<<=1;
              i++;
            }
          if (i <= 31) t.uni[1]|=tag;
          else OVERFLOW;
        }
    }
  else
    { tag=0x1;
      while ((t.uni[0] & tag) == 0x0 && tag != 0x0)
        { t.uni[0]|=tag;
          tag<<=1;
        }
      if (tag != 0x0) t.uni[0]&=~tag;
      else
        { tag=0x1;
          i=1;
          while ((t.uni[1] & tag) == 0x0 && i <= 31)
            { t.uni[1]|=tag;
              tag<<=1;
              i++;
            }
          if (i <= 31) t.uni[1]&=~tag;
          else t.unf=0;
        }
    }
  return t;
}


/*************************************************************************/
/*                              pred.m (MP1)                             */
/*************************************************************************/
#include "mytype.h"
extern plural singleton succ(plural singleton);
plural singleton pred(x)
plural singleton x;
{ plural singleton t;
```

```
  t.unf = -(x.unf);
  t = succ(t);
  t.unf = -(t.unf);
  return t;
}


/*****************************************************************************/
/*                                gadd.m (MP1)                               */
/*****************************************************************************/
#include "mytype.h"
#include <float.h>
#include <math.h>
#define overflow 0
#define underflow 0
extern plural singleton succ(plural singleton);
extern plural singleton pred(plural singleton);
plural intvl gadd(x,y)
plural intvl x,y;
{ plural intvl t1,t2;
  t1.r.unf=x.r.unf+y.r.unf;
  if (overflow) return;
  if (underflow)
    { if (t1.r.unf < 0) t1.r.unf=0;
      else t1.r.unf=DBL_MIN;
    }
  else
    { if (p_fabs(y.r.unf) > p_fabs(x.r.unf))
        { t2=x; x=y; y=t2;}
      t2.r.unf=x.r.unf-t1.r.unf;
      if (overflow)
        { if (y.r.unf < 0) t1.r=succ(t1.r);
        }
      else
        { if (underflow)
            { if (y.r.unf > 0) t1.r=succ(t1.r);
            }
          else
            { t2.r.unf=t2.r.unf+y.r.unf;
              if (t2.r.unf > 0) t1.r=succ(t1.r);
            }
        }
```

```
    }
  t1.l.unf=x.l.unf+y.l.unf;
  if (overflow) return;
  if (underflow)
    { if (t1.l.unf < 0) t1.l.unf = -DBL_MIN;
      else t1.l.unf=0;
    }
  else
    { if (p_fabs(y.l.unf) > p_fabs(x.l.unf))
        { t2=x; x=y; y=t2;}
      t2.l.unf=x.l.unf-t1.l.unf;
      if (overflow)
        { if (y.l.unf > 0) t1.l=pred(t1.l);
        }
      else
        { if (underflow)
            { if (y.l.unf < 0) t1.l=pred(t1.l);
            }
          else
            { t2.l.unf=t2.l.unf+y.l.unf;
              if (t2.l.unf < 0) t1.l=pred(t1.l);
            }
        }
    }
  return t1;
}


/*****************************************************************************/
/*                              gsub.m (MP1)                                 */
/*****************************************************************************/
#include "mytype.h"
extern plural intvl gadd(plural intvl,plural intvl);
plural intvl gsub(x,y)
plural intvl x,y;
{ plural intvl t;
  t.l.unf = -y.r.unf;
  t.r.unf = -y.l.unf;
  return gadd(x,t);
}
```

```
/*****************************************************************************/
/*                              gmul.m (MP1)                                 */
/*****************************************************************************/
#include "mytype.h"
#include <float.h>
#define overflow 0
#define underflow 0
extern plural singleton succ(plural singleton);
extern plural singleton pred(plural singleton);
plural intvl gmul(x,y)
plural intvl x,y;
{ plural double t1l,t2l,t3l,t4l,t1r,t2r,t3r,t4r;
  plural singleton t;
  plural intvl tm;
  t.unf=x.r.unf*y.r.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t1r=DBL_MIN; t1l=0;}
      else {t1r=0; t1l = -DBL_MIN;}
    }
  else {t1r=succ(t).unf; t1l=pred(t).unf;}
  t.unf=x.r.unf*y.l.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t2r=DBL_MIN; t2l=0;}
      else {t2r=0; t2l = -DBL_MIN;}
    }
  else {t2r=succ(t).unf; t2l=pred(t).unf;}
  t.unf=x.l.unf*y.r.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t3r=DBL_MIN; t3l=0;}
      else {t3r=0; t3l = -DBL_MIN;}
    }
  else {t3r=succ(t).unf; t3l=pred(t).unf;}
  t.unf=x.l.unf*y.l.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t4r=DBL_MIN; t4l=0;}
      else {t4r=0; t4l = -DBL_MIN;}
    }
  else {t4r=succ(t).unf; t4l=pred(t).unf;}
```

```
  tm.r.unf=t1r;
  tm.l.unf=t1l;
  if (t2r > tm.r.unf) tm.r.unf=t2r;
  if (t3r > tm.r.unf) tm.r.unf=t3r;
  if (t4r > tm.r.unf) tm.r.unf=t4r;
  if (t2l < tm.l.unf) tm.l.unf=t2l;
  if (t3l < tm.l.unf) tm.l.unf=t3l;
  if (t4l < tm.l.unf) tm.l.unf=t4l;
  return tm;
}


/*****************************************************************************/
/*                                gdiv.m (MP1)                               */
/*****************************************************************************/
#include "mytype.h"
#include <float.h>
#define overflow 0
#define underflow 0
extern plural singleton succ(plural singleton);
extern plural singleton pred(plural singleton);
plural intvl gdiv(x,y)
plural intvl x,y;
{ plural double t1l,t2l,t3l,t4l,t1r,t2r,t3r,t4r;
  plural singleton t;
  plural intvl tm;
  if (y.l.unf*y.r.unf < 0) return;
  t.unf=x.r.unf/y.l.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t1r=DBL_MIN; t1l=0;}
      else {t1r=0; t1l = -DBL_MIN;}
    }
  else {t1r=succ(t).unf; t1l=pred(t).unf;}
  t.unf=x.r.unf/y.r.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t2r=DBL_MIN; t2l=0;}
      else {t2r=0; t2l = -DBL_MIN;}
    }
  else {t2r=succ(t).unf; t2l=pred(t).unf;}
  t.unf=x.l.unf/y.l.unf;
```

```
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t3r=DBL_MIN; t3l=0;}
      else {t3r=0; t3l = -DBL_MIN;}
    }
  else {t3r=succ(t).unf; t3l=pred(t).unf;}
  t.unf=x.l.unf/y.r.unf;
  if (overflow) return;
  if (underflow)
    { if (t.unf > 0) {t4r=DBL_MIN; t4l=0;}
      else {t4r=0; t4l = -DBL_MIN;}
    }
  else {t4r=succ(t).unf; t4l=pred(t).unf;}
  tm.r.unf=t1r;
  tm.l.unf=t1l;
  if (t2r > tm.r.unf) tm.r.unf=t2r;
  if (t3r > tm.r.unf) tm.r.unf=t3r;
  if (t4r > tm.r.unf) tm.r.unf=t4r;
  if (t2l < tm.l.unf) tm.l.unf=t2l;
  if (t3l < tm.l.unf) tm.l.unf=t3l;
  if (t4l < tm.l.unf) tm.l.unf=t4l;
  return tm;
}


/*****************************************************************************/
/*                              gexp.m (MP1)                                 */
/*****************************************************************************/
#include "mytype.h"
#include <stdio.h>
#include <float.h>
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
plural intvl exppos(y)
plural intvl y;
{ plural int i,j;
  plural double length=DBL_MAX;
  plural intvl x,expx,nexterm,dr,total,pre,res,one,result;
  x.l.unf=x.r.unf=y.l.unf;
  one.l.unf=one.r.unf=1.0;
```

```
expx.l.unf=expx.r.unf=1.0;
nexterm=x;
i=0;
while (++i)
  { expx=gadd(expx,nexterm);
    dr.l.unf=dr.r.unf=i+1;
    nexterm=gdiv(nexterm,dr);
    nexterm=gmul(nexterm,x);
    if (x.l.unf < i+1)
      { res=gdiv(x,dr);
        res=gsub(one,res);
        res=gdiv(nexterm,res);
        res.l.unf=0.0;
        total=gadd(expx,res);
        if (total.r.unf-total.l.unf >= length) break;
        else { length=total.r.unf-total.l.unf; pre=total;}
      }
  }
if (y.l.unf == y.r.unf) return pre;
else
  { result.l.unf=pre.l.unf;
    length=DBL_MAX;
    x.l.unf=x.r.unf=y.r.unf;
    expx.l.unf=expx.r.unf=1.0;
    nexterm=x;
    i=0;
    while (++i)
      { expx=gadd(expx,nexterm);
        dr.l.unf=dr.r.unf=i+1;
        nexterm=gdiv(nexterm,dr);
        nexterm=gmul(nexterm,x);
        if (x.l.unf < i+1)
          { res=gdiv(x,dr);
            res=gsub(one,res);
            res=gdiv(nexterm,res);
            res.l.unf=0.0;
            total=gadd(expx,res);
            if (total.r.unf-total.l.unf >= length) break;
            else { length=total.r.unf-total.l.unf; pre=total;}
          }
      }
    result.r.unf=pre.r.unf;
```

```
      return result;
    }
}
plural intvl gexp(y)
plural intvl y;
{ plural intvl x,tmp,tmp1,tmp2,tmp3,one;
  if (y.l.unf >= 0 && y.r.unf >= 0)
    { tmp3=exppos(y);
      if (y.l.unf == 0) tmp3.l.unf=1.0;
      if (y.r.unf == 0) tmp3.r.unf=1.0;
      return tmp3;
    }
  else
    { if (y.l.unf <= 0 && y.r.unf <= 0)
        { x.l.unf = -y.r.unf;
          x.r.unf = -y.l.unf;
          tmp=exppos(x);
          one.l.unf=one.r.unf=1.0;
          tmp3=gdiv(one,tmp);
          if (y.l.unf == 0) tmp3.l.unf=1.0;
          if (y.r.unf == 0) tmp3.r.unf=1.0;
          return tmp3;
        }
      else
        { tmp1.l.unf=0;
          tmp1.r.unf=y.r.unf;
          tmp1=exppos(tmp1);
          tmp2.l.unf=0;
          tmp2.r.unf= -y.l.unf;
          tmp2=exppos(tmp2);
          one.l.unf=one.r.unf=1.0;
          tmp2=gdiv(one,tmp2);
          tmp3.l.unf=tmp2.l.unf;
          tmp3.r.unf=tmp1.r.unf;
          return tmp3;
        }
    }
}


/*****************************************************************************/
/*                              gln.m (MP1)                                  */
```

```
/*************************************************************************/
#include "mytype.h"
#include <stdio.h>
#include <float.h>
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
plural intvl lnpos(y)
plural intvl y;
{ plural int i,j;
  plural double length=DBL_MAX;
  plural intvl x,xsq,lnx,xpower,term,dr,total,pre,con,coe,one,two,result;
  one.l.unf=one.r.unf=1.0;
  x.l.unf=x.r.unf=y.l.unf;
  x=gdiv(gsub(x,one),gadd(x,one));
  xsq=gmul(x,x);
  lnx=x;
  xpower=gmul(xsq,x);
  i=0;
  while (++i)
    { dr.l.unf=dr.r.unf=2*i+1;
      term=gdiv(xpower,dr);
      lnx=gadd(lnx,term);
      xpower=gmul(xpower,xsq);
      dr.l.unf=dr.r.unf=2*i+3;
      coe=gdiv(xpower,dr);
      con=gsub(one,xsq);
      coe=gdiv(coe,con);
      coe.l.unf=0.0;
      total=gadd(lnx,coe);
      if (total.r.unf-total.l.unf >= length) break;
      else { length=total.r.unf-total.l.unf; pre=total;}
    }
  two.l.unf=two.r.unf=2.0;
  if (y.l.unf == y.r.unf) return gmul(two,pre);
  else
   { result.l.unf=gmul(two,pre).l.unf;
     length=DBL_MAX;
     x.l.unf=x.r.unf=y.r.unf;
     x=gdiv(gsub(x,one),gadd(x,one));
     xsq=gmul(x,x);
```

```
      lnx=x;
      xpower=gmul(xsq,x);
      i=0;
      while (++i)
        { dr.l.unf=dr.r.unf=2*i+1;
          term=gdiv(xpower,dr);
          lnx=gadd(lnx,term);
          xpower=gmul(xpower,xsq);
          dr.l.unf=dr.r.unf=2*i+3;
          coe=gdiv(xpower,dr);
          con=gsub(one,xsq);
          coe=gdiv(coe,con);
          coe.l.unf=0.0;
          total=gadd(lnx,coe);
          if (total.r.unf-total.l.unf >= length) break;
          else { length=total.r.unf-total.l.unf; pre=total;}
        }
      result.r.unf=gmul(two,pre).r.unf;
      return result;
    }
}
plural intvl gln(y)
plural intvl y;
{ plural intvl x,tmp1,tmp2,tmp3,one,negone;
  if (y.l.unf <= 0 || y.r.unf <= 0) return;
  if (y.l.unf >= 1 && y.r.unf >= 1)
    { tmp3=lnpos(y);
      if (y.l.unf == 1) tmp3.l.unf=0.0;
      if (y.r.unf == 1) tmp3.r.unf=0.0;
      return tmp3;
    }
  else
    { if (y.l.unf <= 1 && y.r.unf <= 1)
        { one.l.unf=one.r.unf=1.0;
          x=gdiv(one,y);
          tmp3=lnpos(x);
          negone.l.unf=negone.r.unf= -1.0;
          tmp3=gmul(negone,tmp3);
          if (y.l.unf == 1) tmp3.l.unf=0.0;
          if (y.r.unf == 1) tmp3.r.unf=0.0;
          return tmp3;
        }
```

```
      else
        { tmp1.l.unf=1;
          tmp1.r.unf=y.r.unf;
          tmp1=lnpos(tmp1);
          one.l.unf=one.r.unf=1.0;
          tmp2.r.unf=1.0;
          tmp2.l.unf=y.l.unf;
          tmp2=gdiv(one,tmp2);
          tmp2=lnpos(tmp2);
          negone.l.unf=negone.r.unf= -1.0;
          tmp2=gmul(negone,tmp2);
          tmp3.l.unf=tmp2.l.unf;
          tmp3.r.unf=tmp1.r.unf;
          return tmp3;
        }
    }
}


/*************************************************************************/
/*                          feupdown.c (MP1)                             */
/*************************************************************************/
/* This program does the F.E. computation for mulnor_wrap.m etc.      */
/* The front end is just a DEC 5000/240, which has IEEE rounding mode */
/* control. We don't have to bother using the Clemmesen algorithms.   */
/*************************************************************************/

#include <stdio.h>
#include "/usr/include/mips/fpu.h"

#define maxn 10
#define maxintv 10

int ffn,ffm[maxn+1];
double fa[maxn+1][maxintv+1];
struct {double l,r;} ffc[maxn+1][maxintv+1],ffh[maxn+1][maxintv+1],
                     ffx[maxn+1][maxintv+1];

feupdown()
   {
    int sw,mask=3,twobits,fi,fj;
/* set rounding mode to round to plus infinity (low order bits=2) */
```

```
    sw=get_fpc_csr();

    /* use mask to extract two low order bits from the status register */
    twobits=sw & mask;  /* twobits is now = 0 or 1 or 2 or 3 */

    /* adjust the two low order bits to equal 2, then set the register */
    switch(twobits)
      {
      case 0:
        set_fpc_csr(sw+2);
        break;
      case 1:
        set_fpc_csr(sw+1);
        break;
      case 3:
        set_fpc_csr(sw-1);
        break;
      default:
        break;
      }
    for (fi=1;fi<=ffn;fi++)
     for (fj=1;fj<=ffm[fi];fj++)
       { ffc[fi][fj].r=(fa[fi][fj-1]+fa[fi][fj])/2.0;
         ffh[fi][fj].r=(fa[fi][fj]-fa[fi][fj-1])/2.0;
         ffx[fi][fj].r=fa[fi][fj];
       }

/* set rounding mode to round to minus infinity (low order bits=3) */
    sw=get_fpc_csr();

/* use mask to extract the two low order bits from the status register */
    twobits=sw & mask;  /* twobits is now = to 0 or 1 or 2 or 3 */

/* adjust the two low order bits to equal 3, then set the register */
    switch(twobits)
      {
      case 0:
        set_fpc_csr(sw+3);
        break;
      case 1:
        set_fpc_csr(sw+2);
        break;
```

```
      case 2:
        set_fpc_csr(sw+1);
        break;
      default:
        break;
      }
    for (fi=1;fi<=ffn;fi++)
     for (fj=1;fj<=ffm[fi];fj++)
       { ffc[fi][fj].l=(fa[fi][fj-1]+fa[fi][fj])/2.0;
         ffh[fi][fj].l=(fa[fi][fj]-fa[fi][fj-1])/2.0;
         ffx[fi][fj].l=fa[fi][fj-1];
       }

/* move back to round to nearest */
    sw=get_fpc_csr();

/* use mask to extract the two low order bits from the status register */
    twobits=sw & mask;  /* twobits is now = to 0 or 1 or 2 or 3 */

/* adjust the low order bits to equal 0, then set the register */
    switch(twobits)
      {
      case 1:
        set_fpc_csr(sw-1);
        break;
      case 2:
        set_fpc_csr(sw-2);
        break;
      case 3:
        set_fpc_csr(sw-3);
        break;
      default:
        break;
      }
   }


/*****************************************************************************/
/*                          mytype.h (MP2)                                   */
/*****************************************************************************/
typedef struct { double l;
                 double r;
```

```
} intvl;

/*****************************************************************************/
/*                              gadd.m (MP2)                                 */
/*****************************************************************************/
#include "mytype.h"
#include <fenv.h>
#include <float.h>
#include <stdio.h>
#define INF DBL_MAX
#define MINF -DBL_MAX
plural intvl gadd(x,y)
plural intvl x,y;
{ plural intvl z;
  if ((x.r==INF && y.r==MINF) || (x.r==MINF && y.r==INF))
    { p_printf("Invilid addition!\n");
      return z;
    }
  else
    { fesetround(FE_UPWARD);
      if (x.r==INF || y.r==INF) z.r=INF;
      else
        { if (x.r==MINF || y.r==MINF) z.r=MINF;
          else z.r=x.r+y.r;
        }
    }
  if ((x.l==INF && y.l==MINF) || (x.l==MINF && y.l==INF))
    { p_printf("Invilid addition!\n");
      return z;
    }
  else
    { fesetround(FE_DOWNWARD);
      if (x.l==INF || y.l==INF) z.l=INF;
      else
        { if (x.l==MINF || y.l==MINF) z.l=MINF;
          else z.l=x.l+y.l;
        }
    }
  fesetround(FE_TONEAREST);
  return z;
}
```

```
/*****************************************************************/
/*                          gsub.m (MP2)                         */
/*****************************************************************/
#include "mytype.h"
extern plural intvl gadd(plural intvl,plural intvl);
plural intvl gsub(x,y)
plural intvl x,y;
{ plural intvl t;
  t.l = -y.r;
  t.r = -y.l;
  return gadd(x,t);
}


/*****************************************************************/
/*                          gmul.m (MP2)                         */
/*****************************************************************/
#include "mytype.h"
#include <fenv.h>
#include <float.h>
#include <stdio.h>
#define INF DBL_MAX
#define MINF -DBL_MAX
plural intvl gmul(x,y)
plural intvl x,y;
{ plural intvl z;
  plural double p1,p2,p3,p4;
  plural double q1,q2,q3,q4;
  if (((x.l==INF || x.l==MINF) && y.l==0) || ((y.l==INF || y.l==MINF)
                                                  && x.l==0))
    { p_printf("Invalid Multiplication!\n");
      return z;
    }
  else
    { if ((x.l==INF && y.l>0) || (x.l==MINF && y.l<0) ||
                 (y.l==INF && x.l>0) || (y.l==MINF && x.l<0)) p1=q1=INF;
      else if ((x.l==INF && y.l<0) || (x.l==MINF && y.l>0) ||
                (y.l==INF && x.l<0) || (y.l==MINF && x.l>0)) p1=q1=MINF;
           else
             { fesetround(FE_DOWNWARD);
```

```
            p1=x.l*y.l;
            fesetround(FE_UPWARD);
            q1=x.l*y.l;
          }
  }
if (((x.l==INF || x.l==MINF) && y.r==0) || ((y.r==INF || y.r==MINF)
                                                    && x.l==0))
  { p_printf("Invalid Multiplication!\n");
    return z;
  }
else
  { if ((x.l==INF && y.r>0) || (x.l==MINF && y.r<0) ||
              (y.r==INF && x.l>0) || (y.r==MINF && x.l<0)) p2=q2=INF;
    else if ((x.l==INF && y.r<0) || (x.l==MINF && y.r>0) ||
              (y.r==INF && x.l<0) || (y.r==MINF && x.l>0)) p2=q2=MINF;
         else
           { fesetround(FE_DOWNWARD);
             p2=x.l*y.r;
             fesetround(FE_UPWARD);
             q2=x.l*y.r;
           }
  }
if (((x.r==INF || x.r==MINF) && y.l==0) || ((y.l==INF || y.l==MINF)
                                                    && x.r==0))
  { p_printf("Invalid Multiplication!\n");
    return z;
  }
else
  { if ((x.r==INF && y.l>0) || (x.r==MINF && y.l<0) ||
              (y.l==INF && x.r>0) || (y.l==MINF && x.r<0)) p3=q3=INF;
    else if ((x.r==INF && y.l<0) || (x.r==MINF && y.l>0) ||
              (y.l==INF && x.r<0) || (y.l==MINF && x.r>0)) p3=q3=MINF;
         else
           { fesetround(FE_DOWNWARD);
             p3=x.r*y.l;
             fesetround(FE_UPWARD);
             q3=x.r*y.l;
           }
  }
if (((x.r==INF || x.r==MINF) && y.r==0) || ((y.r==INF || y.r==MINF)
                                                    && x.r==0))
  { p_printf("Invalid Multiplication!\n");
```

```
      return z;
    }
  else
    { if ((x.r==INF && y.r>0) || (x.r==MINF && y.r<0) ||
                (y.r==INF && x.r>0) || (y.r==MINF && x.r<0)) p4=q4=INF;
      else if ((x.r==INF && y.r<0) || (x.r==MINF && y.r>0) ||
                (y.r==INF && x.r<0) || (y.r==MINF && x.r>0)) p4=q4=MINF;
           else
             { fesetround(FE_DOWNWARD);
               p4=x.r*y.r;
               fesetround(FE_UPWARD);
               q4=x.r*y.r;
             }
    }
  z.l=p1;
  z.r=q1;
  if (z.l>p2) z.l=p2;
  if (z.l>p3) z.l=p3;
  if (z.l>p4) z.l=p4;
  if (z.r<q2) z.r=q2;
  if (z.r<q3) z.r=q3;
  if (z.r<q4) z.r=q4;
  fesetround(FE_TONEAREST);
  return z;
}


/*****************************************************************************/
/*                              gdiv.m (MP2)                                 */
/*****************************************************************************/
#include "mytype.h"
#include <fenv.h>
#include <float.h>
#include <stdio.h>
#define INF DBL_MAX
#define MINF -DBL_MAX
extern plural intvl gmul(plural intvl,plural intvl);
plural intvl grec(y)
plural intvl y;
{ plural intvl z;
  if ((y.l==MINF && y.r==MINF) || (y.l==INF && y.r==INF)) z.l=z.r=0;
  else if (y.l==MINF && y.r<=0)
```

```
        { z.r=0;
          if (y.r==0) z.l=MINF;
          else
            { fesetround(FE_DOWNWARD);
              z.l=1.0/y.r;
            }
        }
      else if (y.r==INF && y.l>=0)
             { z.l=0;
               if (y.l==0) z.r=INF;
               else
                 { fesetround(FE_UPWARD);
                   z.r=1.0/y.l;
                 }
             }
           else if (y.l<0 && y.r>0) {z.l=MINF; z.r=INF;}
                else if (y.l==0 && y.r==0) z.l=z.r=INF;
                     else if (y.l==0 && y.r>0)
                            { z.r=INF;
                              fesetround(FE_DOWNWARD);
                              z.l=1.0/y.r;
                            }
                          else if (y.l<0 && y.r==0)
                                 { z.l=MINF;
                                   fesetround(FE_UPWARD);
                                   z.r=1.0/y.l;
                                 }
                               else
                                 { fesetround(FE_DOWNWARD);
                                   z.l=1.0/y.r;
                                   fesetround(FE_UPWARD);
                                   z.r=1.0/y.l;
                                 }
  fesetround(FE_TONEAREST);
  return z;
}
plural intvl gdiv(x,y)
plural intvl x,y;
{ return gmul(x,grec(y));
}
```

```
/*****************************************************************************/
/*                              gsqu.m (MP2)                                 */
/*****************************************************************************/
#include "mytype.h"
extern plural intvl gmul(plural intvl,plural intvl);
plural intvl gsqu(x)
plural intvl x;
{ plural intvl tmp;
  tmp=gmul(x,x);
  if (x.l<0 && x.r>0) tmp.l=0;
  return tmp;
}


/*****************************************************************************/
/*                              gsqt.m (MP2)                                 */
/*****************************************************************************/
#include "mytype.h"
#include <stdio.h>
#include <float.h>
#include <math.h>
#define INF DBL_MAX
#define MINF -DBL_MAX
#define eps 0.01
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
extern plural intvl gsqu(plural intvl);
plural intvl gsqt(x)
plural intvl x;
{ plural intvl sqt,endx,y,z,t1,t2,two,mid;
  plural double length;
  plural int goon;
  if (x.l<0)
    { p_printf("Square root of a negative?\n");
      return y;
    }
  two.l=two.r=2;
  length=DBL_MAX;
  if (x.l==0) sqt.l=0;
  else if (x.l==INF) sqt.l=INF;
```

```
      else
        { endx.l=endx.r=x.l;;
          y.l=p_sqrt(x.l)-eps;
          y.r=y.l+2*eps;
          y.l=y.l>0?y.l:DBL_MIN;
          while (y.l*y.l>x.l) y.l-=eps;
          while (y.r*y.r<x.l) y.r+=eps;
          goon=1;
          while (goon)
            { t1.l=t1.r=y.l;
              t2.l=t2.r=y.r;
              mid=gdiv(gadd(t1,t2),two);
              z=gsub(mid,gdiv(gsub(gsqu(mid),endx),gmul(two,y)));
              y.l=y.l>z.l?y.l:z.l;
              y.r=y.r<z.r?y.r:z.r;
              if (y.r-y.l>=length) goon=0;
              else length=y.r-y.l;
            }
          sqt.l=y.l;
        }
length=DBL_MAX;
if (x.r==0) sqt.r=0;
else if (x.r==INF) sqt.r=INF;
      else
        { endx.l=endx.r=x.r;;
          y.l=p_sqrt(x.r)-eps;
          y.r=y.l+2*eps;
          y.l=y.l>0?y.l:DBL_MIN;
          while (y.l*y.l>x.r) y.l-=eps;
          while (y.r*y.r<x.r) y.r+=eps;
          goon=1;
          while (goon)
            { t1.l=t1.r=y.l;
              t2.l=t2.r=y.r;
              mid=gdiv(gadd(t1,t2),two);
              z=gsub(mid,gdiv(gsub(gsqu(mid),endx),gmul(two,y)));
              y.l=y.l>z.l?y.l:z.l;
              y.r=y.r<z.r?y.r:z.r;
              if (y.r-y.l>=length) goon=0;
              else length=y.r-y.l;
            }
          sqt.r=y.r;
```

```
        }
  return sqt;
}


/*************************************************************************/
/*                              gexp.m (MP2)                             */
/*************************************************************************/
#include "mytype.h"
#include <stdio.h>
#include <float.h>
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
plural intvl exppos(y)
plural intvl y;
{ plural int i,j;
  plural double length=DBL_MAX;
  plural intvl x,expx,nexterm,dr,total,pre,res,one,result;
  x.l=x.r=y.l;
  one.l=one.r=1.0;
  expx.l=expx.r=1.0;
  nexterm=x;
  i=0;
  while (++i)
    { expx=gadd(expx,nexterm);
      dr.l=dr.r=i+1;
      nexterm=gdiv(nexterm,dr);
      nexterm=gmul(nexterm,x);
      if (x.l < i)
        { res=gdiv(x,dr);
          res=gsub(one,res);
          res=gdiv(nexterm,res);
          res.l=0.0;
          total=gadd(expx,res);
          if (total.r-total.l >= length) break;
          else { length=total.r-total.l; pre=total;}
        }
    }
  if (y.l == y.r) return pre;
  else
```

```
    { result.l=pre.l;
      length=DBL_MAX;
      x.l=x.r=y.r;
      expx.l=expx.r=1.0;
      nexterm=x;
      i=0;
      while (++i)
        { expx=gadd(expx,nexterm);
          dr.l=dr.r=i+1;
          nexterm=gdiv(nexterm,dr);
          nexterm=gmul(nexterm,x);
          if (x.l < i)
            { res=gdiv(x,dr);
              res=gsub(one,res);
              res=gdiv(nexterm,res);
              res.l=0.0;
              total=gadd(expx,res);
              if (total.r-total.l >= length) break;
              else { length=total.r-total.l; pre=total;}
            }
        }
      result.r=pre.r;
      return result;
    }
}
plural intvl gexp(y)
plural intvl y;
{ plural intvl x,tmp,tmp1,tmp2,tmp3,one;
  if (y.l >= 0 && y.r >= 0)
    { tmp3=exppos(y);
      if (y.l == 0) tmp3.l=1.0;
      if (y.r == 0) tmp3.r=1.0;
      return tmp3;
    }
  else
    { if (y.l <= 0 && y.r <= 0)
        { x.l = -y.r;
          x.r = -y.l;
          tmp=exppos(x);
          one.l=one.r=1.0;
          tmp3=gdiv(one,tmp);
          if (y.l == 0) tmp3.l=1.0;
```

```
            if (y.r == 0) tmp3.r=1.0;
            return tmp3;
          }
        else
          { tmp1.l=0;
            tmp1.r=y.r;
            tmp1=exppos(tmp1);
            tmp2.l=0;
            tmp2.r= -y.l;
            tmp2=exppos(tmp2);
            one.l=one.r=1.0;
            tmp2=gdiv(one,tmp2);
            tmp3.l=tmp2.l;
            tmp3.r=tmp1.r;
            return tmp3;
          }
      }
}
```

APPENDIX B. MASPAR SOURCE CODES FOR SURVEY SAMPLING ANALYSIS

```
/*************************************************************************/
/*                              bigtotal.c                               */
/*************************************************************************/
/* Youngs and Cramer's algorithm. Technometrics vol 13 number 3.    */
/* 71/8 p.657. algorithm S7. May refer to /home/ouhong/carp/total.m */
/* for variable settings. They are very similar.                    */
/* Can deal with any # of variables. The program splits the problem */
/* and handles a maxvar*maxvar matrix at a time.                    */
/* running time is 20 sec vs. 25 min (MasPar vs. 386)               */
/* if strata begins from 1, j=0 should be changed to                */
/* j=1 (in file ratein.m).                                          */
/*************************************************************************/
#include <stdio.h>
#include <malloc.h>
#define maxvar 50
#define mpsize 16384
#define nxproc 128
#define nyproc 128
extern void ratein(),ttlonly(),covpart();
extern int nstrt,cov,penum,nrow,ncol,actvar1[maxvar],actvar2[maxvar],
       lastid;
extern double sneak[2],v[maxvar][maxvar],leftover[maxvar][maxvar],
       ttl[maxvar],lastx1[maxvar],lasty1[maxvar],lastx2[maxvar],
       lasty2[maxvar],lastvar[2];
extern double var1[maxvar+3],var2[maxvar+3];
main()
{ FILE *fd;
  int i,i1,i2,j,j1,j2,k1,k2,l1,l2,m1,m2,m3,fecov,fepenum,fe_num_var,
      fe_num_strt,*feactvar,felastid;
```

```
char data[30];
double *varpool,*fevar[mpsize],*fettl,**fev,**feleftover,fesneak[2];
double *felastx1,*felasty1,*felastx2,*felasty2,felastvar[2];
printf("Input data file name : ");
scanf("%s",data);
printf("Input number of variables in the data file : ");
                    /* including strata,cluster and weight */
scanf("%d",&fe_num_var);
printf("Input number of strata (strata has to start from 0) : ");
scanf("%d",&fe_num_strt);
printf("Do you want the Var-Cov matrix ? (1-yes 0-no) ");
scanf("%d",&fecov);
feactvar=(int *)malloc(sizeof(int)*fe_num_var);
printf("Do you want TOTAL for which variables? (1-yes 0-no)\n");
for (i=3;i<=fe_num_var-1;i++)
 { printf("variable #%d : ",i+1);
   scanf("%d",&feactvar[i]);
 }
copyOut(&fe_num_strt,&nstrt,sizeof(int));
callRequest(ratein,0); /* get the rates */
fd=fopen(data,"r");
varpool=(double *)malloc(sizeof(double)*maxvar*mpsize);
             /* used as a buffer to put the fe variables in */
             /* to transfer to pe or acu                    */
for (i=0;i<=mpsize-1;i++)
  fevar[i]=(double *)malloc(sizeof(double)*fe_num_var);
fettl=(double *)malloc(sizeof(double)*fe_num_var);
fev=(double **)malloc(sizeof(double *)*fe_num_var);
feleftover=(double **)malloc(sizeof(double *)*fe_num_var);
felastx1=(double *)malloc(sizeof(double)*fe_num_var);
felasty1=(double *)malloc(sizeof(double)*fe_num_var);
felastx2=(double *)malloc(sizeof(double)*fe_num_var);
felasty2=(double *)malloc(sizeof(double)*fe_num_var);
for (i=0;i<=fe_num_var-1;i++)
  { fev[i]=(double *)malloc(sizeof(double)*(i+1));
      /* make triangle matrix since the cov matrix is symmetric */
    feleftover[i]=(double *)malloc(sizeof(double)*(i+1));
  }
for (i=0;i<=fe_num_var-1;i++)
  { fettl[i]=0;
    for (j=0;j<=i;j++)
      { fev[i][j]=0;
```

```
          feleftover[i][j]=0;
        }
    }
  copyOut(&fecov,&cov,sizeof(int));
  felastid=0;
  felastvar[0]=felastvar[1]= -1;
  fscanf(fd,"%lf",&fesneak[0]);
      /* sneak[0,1] are the strata and cluster on the first pe node */
      /* of the next round. Used to detect if end of strata or      */
      /* cluster is reached                                         */
  fscanf(fd,"%lf",&fesneak[1]);
  while (! feof(fd))
   { fepenum=mpsize; /* # of active pe nodes for this round */
     for (i=0;i<=mpsize-1;i++)
      if (! feof(fd))
        { if (i==0)
            { fevar[0][0]=fesneak[0];
              fevar[0][1]=fesneak[1];
              fesneak[0]=fesneak[1]= -1;
              for (j=2;j<=fe_num_var-1;j++)
                 fscanf(fd,"%lf",&fevar[0][j]);
            }
          else
            for (j=0;j<=fe_num_var-1;j++)
               fscanf(fd,"%lf",&fevar[i][j]);
        }
      else
        { fepenum=i-1; /* should be i logically, but the program */
                       /* always reads for one more line. Not    */
                       /* sure why(the \n at the end?).          */
          break;
        }
     if (! feof(fd))
/* fesneak are next round's strata and cluster # on the first */
/* active pe node. By looking at fesneak we will know if this */
/* round at the last active pe node we are at the end of a    */
/* strata or a cluster.                                       */
       { fscanf(fd,"%lf",&fesneak[0]);
         fscanf(fd,"%lf",&fesneak[1]);
       }
     j=0;
     for (i=0;i<fepenum;i++)
```

```
      { varpool[j++]=fevar[i][0];
        varpool[j++]=fevar[i][1];
        varpool[j++]=fevar[i][2];
      }

/* strata,cluster and weight are put in their positions.        */
/* var1[0,1,2] are these info. var2[0,1,2] are not used at all. */
/* Whenever we want to use, say, var2[1] we just use var1[1].   */
/* They are always the same.                                    */
     blockOut(varpool,var1,0,0,nxproc,nyproc,sizeof(double)*3);
     copyOut(fesneak,sneak,sizeof(double)*2);
     copyOut(&fepenum,&penum,sizeof(int));
     j1=2;
/* i1,var1 etc, the "1" variables are for the rows of the cov matrix */
/* i2,var2 etc, the "2" variables are for the cols of the cov matrix */
     while (j1<fe_num_var-1)
       { i1=j1+1;
         j1=i1+maxvar-1;
         j1=j1>fe_num_var-1?fe_num_var-1:j1;
/* Deal with variables i1 to j1 for row variables. */
         m1=0;
         for (k1=0;k1<fepenum;k1++)
          for (l1=i1;l1<=j1;l1++)
            varpool[m1++]=fevar[k1][l1];
         blockOut(varpool,&var1[3],0,0,nxproc,nyproc,
                                     sizeof(double)*(j1-i1+1));
/* starts from 3 because 0,1,2 are strata,cluster and weight already */
         copyOut(&feactvar[i1],actvar1,sizeof(int)*(j1-i1+1));
         m1=j1-i1+1; /* # of row variables for this round */
         copyOut(&m1,&nrow,sizeof(int));
         copyOut(&fettl[i1],ttl,sizeof(double)*m1);
                             /* send the total in order to update */
         callRequest(ttlonly,0); /* update the total */
         copyIn(ttl,&fettl[i1],sizeof(double)*m1);
                                     /* send the updated total back */
         j2=2;
         while (j2<j1)
           { i2=j2+1;
             j2=i2+maxvar-1;
             j2=j2>j1?j1:j2;
/* Deal with variables i2 to j2 for column variables. */
             m2=0;
```

```
for (k2=0;k2<fepenum;k2++)
 for (l2=i2;l2<=j2;l2++)
   varpool[m2++]=fevar[k2][l2];
blockOut(varpool,&var2[3],0,0,nxproc,nyproc,
                              sizeof(double)*(j2-i2+1));
copyOut(&feactvar[i2],actvar2,sizeof(int)*(j2-i2+1));
copyOut(&felastx1[i1],lastx1,sizeof(double)*m1);
copyOut(&felasty1[i1],lasty1,sizeof(double)*m1);
m2=j2-i2+1;
copyOut(&m2,&ncol,sizeof(int));
copyOut(&felastx2[i2],lastx2,sizeof(double)*m2);
copyOut(&felasty2[i2],lasty2,sizeof(double)*m2);
m3=0;
for (k2=i1;k2<=i1+maxvar-1;k2++)
 for (l2=i2;l2<=i2+maxvar-1;l2++)
   if (k2>j1 || l2>j2) varpool[m3++]=0;
   else varpool[m3++]=k2>=l2?fev[k2][l2]:fev[l2][k2];
copyOut(varpool,v,sizeof(double)*maxvar*maxvar);
                        /* v is the cov matrix on pe */
m3=0;
for (k2=i1;k2<=i1+maxvar-1;k2++)
 for (l2=i2;l2<=i2+maxvar-1;l2++)
   if (k2>j1 || l2>j2) varpool[m3++]=0;
   else varpool[m3++]=k2>=l2?feleftover[k2][l2]:
                                  feleftover[l2][k2];
copyOut(varpool,leftover,sizeof(double)*maxvar*maxvar);
              /* leftover is use by covpart subroutine */
copyOut(&felastid,&lastid,sizeof(int));
copyOut(felastvar,lastvar,sizeof(double)*2);
callRequest(covpart,0);
if (j1 >= fe_num_var-1)
 { copyIn(lasty2,&felasty2[i2],sizeof(double)*m2);
   copyIn(lastx2,&felastx2[i2],sizeof(double)*m2);
 }
copyIn(leftover,varpool,sizeof(double)*maxvar*maxvar);
              /* copy back the updated leftover matrix */
m3=0;
for (k2=i1;k2<=i1+maxvar-1;k2++)
 for (l2=i2;l2<=i2+maxvar-1;l2++)
   if (k2>j1 || l2>j2 || k2<l2) m3++;
   else feleftover[k2][l2]=varpool[m3++];
copyIn(v,varpool,sizeof(double)*maxvar*maxvar);
```

```
                                    /* copy back the updated v matrix */
            m3=0;
            for (k2=i1;k2<=i1+maxvar-1;k2++)
             for (l2=i2;l2<=i2+maxvar-1;l2++)
               if (k2>j1 || l2>j2 || k2<l2) m3++;
               else fev[k2][l2]=varpool[m3++];
          }
        copyIn(lasty1,&felasty1[i1],sizeof(double)*m1);
        copyIn(lastx1,&felastx1[i1],sizeof(double)*m1);
       }
     copyIn(lastvar,felastvar,sizeof(double)*2);
     copyIn(&lastid,&felastid,sizeof(int));
   }
  printf("Total :\n");
  for (i=0;i<=fe_num_var-1;i++)
    if (feactvar[i]==1) printf("variable #%d : %10.4f\n",i+1,fettl[i]);
  if (fecov==1)
    { printf("Var-Cov :\n");
      for (i=3;i<=fe_num_var-1;i++)
        if (feactvar[i]==1)
          { for (j=3;j<=fe_num_var-1;j++)
             if (feactvar[j]==1)
               { if (i>=j) printf("%24.4f",fev[i][j]);
                 else printf("%24.4f",fev[j][i]);
               }
            printf("\n");
          }
    }
}


/*****************************************************************************/
/*                              covpart.m                                    */
/*****************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpl.h>
#include <mp_libc.h>
#include <ppeio.h>
#define maxvar 50
```

```
extern int selectOne();
extern int selectFirst();
extern plural int strata_flag,cluster_flag;
extern plural double var1[maxvar+3],x1[maxvar];
extern int penum,nrow,actvar1[maxvar];
extern double *acurate;
visible int actvar2[maxvar],ncol,lastid;
visible double v[maxvar][maxvar],leftover[maxvar][maxvar],lastvar[2],
     lastx1[maxvar],lasty1[maxvar],lastx2[maxvar],lasty2[maxvar],sneak[2];
visible plural double var2[maxvar+3],x2[maxvar];
visible void covpart()
{ int j,l,target,fst,substop,stophere,ind=0;
  plural int id,tag;
  plural double rate,y1[maxvar],y2[maxvar],perstrata;
  double temp[maxvar];

  if (iproc<penum) /* only penum pe nodes are used in this round */
   { for (j=0;j<=ncol-1;j++)
       if (actvar2[j]==1)
         { x2[j]=var1[2]*var2[j+3];
           x2[j]=scanAddd(x2[j],cluster_flag);
                       /* values on end of clusters, these are */
                       /* individual x's in the formula        */
         }
     if (cluster_flag==1)
      { id=1;
        if (proc[0].var1[0]!=lastvar[0])
                    /* if last round we happen to encounter the */
                    /* end of a strata at the last active pe    */
          { for (j=0;j<=nrow-1;j++)
              if (actvar1[j]==1) y1[j]=scanAddd(x1[j],strata_flag);
            for (j=0;j<=ncol-1;j++)
              if (actvar2[j]==1) y2[j]=scanAddd(x2[j],strata_flag);
          }
        else
          { fst=selectFirst();
            ind=1;
            if (proc[0].var1[1]==lastvar[1])
                        /* if neither strata end or cluster end was */
                        /* encountered last time                    */
              { proc[fst].id=lastid;
                for (j=0;j<=nrow-1;j++)
```

```
          if (actvar1[j]==1)
            { temp[j]=proc[fst].x1[j];
              proc[fst].x1[j]+=lasty1[j];
              y1[j]=scanAddd(x1[j],strata_flag);
                 /* adjust the values because part of the   */
                 /* cluster was in last round's computation */
              proc[fst].x1[j]=proc[fst].x1[j]-lasty1[j]+lastx1[j];
            }
        for (j=0;j<=ncol-1;j++)
          if (actvar2[j]==1)
            { proc[fst].x2[j]+=lasty2[j];
              y2[j]=scanAddd(x2[j],strata_flag);
              proc[fst].x2[j]=proc[fst].x2[j]-lasty2[j]+lastx2[j];
            }
      }
    else /* if same strata but different cluster between the */
         /* first pe this round and last pe last round       */
      { proc[fst].id=lastid+1;
        for (j=0;j<=nrow-1;j++)
          if (actvar1[j]==1)
            { temp[j]=proc[fst].x1[j];
              proc[fst].x1[j]+=lasty1[j];
              y1[j]=scanAddd(x1[j],strata_flag);
              proc[fst].x1[j]-=lasty1[j];
                           /* adjust differently from above */
            }
        for (j=0;j<=ncol-1;j++)
          if (actvar2[j]==1)
            { proc[fst].x2[j]+=lasty2[j];
              y2[j]=scanAddd(x2[j],strata_flag);
              proc[fst].x2[j]-=lasty2[j];
            }
      }
  }
if (strata_flag==1) tag=1;
while (tag==1)
 { target=selectOne();
   proc[target].rate=acurate[(int)proc[target].var1[0]];
   proc[target].tag=0;
 }
id=scanAdd32(id,strata_flag);
                  /* the order # of cluster in its strata */
```

```
   all { if (proc[nproc-1].var1[0]!=sneak[0]) stophere=nproc-1;
         else if (var1[0]==proc[nproc-1].var1[0])
                stophere=selectFirst()-1;
         if (proc[nproc-1].var1[0]==sneak[0] &&
                              proc[nproc-1].var1[1]==sneak[1])
          { if (var1[0]==proc[nproc-1].var1[0] &&
                             var1[1]==proc[nproc-1].var1[1])
              substop=selectFirst()-1;
          }
       }
   if (penum<nproc) stophere=penum-1;
              /* stophere is to which point we update v matrix */
   if (id>1)
     for (j=0;j<=nrow-1;j++)
      for (l=0;l<=ncol-1;l++)
        if (actvar1[j]==1 && actvar2[l]==1)
          { perstrata=scanAddd((id*x1[j]-y1[j])*(id*x2[l]-y2[l])/
                                 id/(id-1),strata_flag);
                    /* the cov value at the end of each strata */
            if (strata_flag==1)
              proc[selectFirst()].perstrata+=leftover[j][l];
                          /* update the first strata because it */
                          /* has sth to do with the last round  */
            if (iproc<=stophere && strata_flag==1)
              v[j][l]+=reduceAddd(id*(1-rate)/(id-1)*perstrata);
                                                /*update v matrix */
            if (proc[nproc-1].var1[0]!=sneak[0]) leftover[j][l]=0;
                     /* record the adjust value for next round */
            else if (proc[nproc-1].var1[1]==sneak[1])
                   leftover[j][l]=proc[substop].perstrata;
                 else leftover[j][l]=proc[nproc-1].perstrata;
          }
 }
lastid=proc[nproc-1].id;
lastvar[0]=proc[nproc-1].var1[0];
lastvar[1]=proc[nproc-1].var1[1];
for (j=0;j<=nrow-1;j++) if (actvar1[j]==1)
                          { lasty1[j]=proc[nproc-1].y1[j];
                            lastx1[j]=proc[nproc-1].x1[j];
                          }
for (j=0;j<=ncol-1;j++) if (actvar2[j]==1)
                          { lasty2[j]=proc[nproc-1].y2[j];
```

```
                                lastx2[j]=proc[nproc-1].x2[j];
                              }
   }
  if (ind==1)
    for (j=0;j<=nrow-1;j++) if (actvar1[j]==1) proc[fst].x1[j]=temp[j];
}


/*************************************************************************/
/*                              ratein.m                                 */
/*************************************************************************/
#include <stdlib.h>
#include <stdio.h>
visible int nstrt;
double *acurate;
visible void ratein()
{ FILE *fr;
  char ratefile[30];
  int j;

  printf("Input rate file name : ");
  scanf("%s",ratefile);
  fr=fopen(ratefile,"r");
  acurate=(double *)malloc(sizeof(double)*nstrt);
  j=0;
/* here change to j=1 if strata begins from 1 */
  while (! feof(fr)) fscanf(fr,"%lf",&acurate[j++]);
}


/*************************************************************************/
/*                              ttlonly.m                                */
/*************************************************************************/
#include <mpl.h>
#define maxvar 50
visible int penum,actvar1[maxvar],nrow,cov;
visible double ttl[maxvar];
visible plural double var1[maxvar+3],x1[maxvar];
plural int strata_flag,cluster_flag;
visible void ttlonly()
{ int j;
  if (iproc<penum)
```

```
  { for (j=0;j<=nrow-1;j++) if (actvar1[j]==1)
                              ttl[j]+=reduceAddd(var1[2]*var1[j+3]);
    if (cov==1)
      { if (var1[0]==router[iproc+1].var1[0]) strata_flag=0;
          else strata_flag=1;
        if (var1[1]==router[iproc+1].var1[1] &&
                                    var1[0]==router[iproc+1].var1[0])
          cluster_flag=0;
        else cluster_flag=1;
        if (iproc==penum-1) strata_flag=cluster_flag=1;
        for (j=0;j<=nrow-1;j++)
          if (actvar1[j]==1)
            { x1[j]=var1[2]*var1[j+3];
              x1[j]=scanAddd(x1[j],cluster_flag);
            }
      }
  }
}
```

APPENDIX C. MASPAR SOURCE CODES FOR MULTIVARIATE INTEGRATION

We include here only the program for multivariate Normal integration.

```
/*************************************************************************/
/*                          mulnor_wrap_recur.m                          */
/*************************************************************************/
/* Compute multivariate normal integration over a rectangle. Required  */
/* parameters:                                                          */
/*             n : dimension (<=10)                                     */
/*             ai,bi i=1..n : interval on each dimension (length<=19)   */
/*             t[i][j] : elements of sigma inverse.                     */
/* A subregion with each dimension of length <=1.9 is computed at a     */
/* time. There is a limit of only one subregion. (Can modify to accept */
/* more subregions and store them on F.E., pass one to PE array at a    */
/* time. This way we can eliminate the dimentions for subregions of     */
/* arrays of plural type. Have not done this!) For each n, a            */
/* multivariate Taylor expansion of fmaxk terms is used. For each of    */
/* k=0..fmaxk-1, the result is printed out.                             */
/*************************************************************************/
/* If k reaches beyond fmaxk, there will be too many f values to store */
/* in the limited pe memory. We just store f values for k upto fmaxk.  */
/* For k > fmaxk, the f value is computed recursively, ie, k uses f    */
/* values in k-1 and k-2 families, the k-1 f value needs f values in   */
/* k-2 and k-3 families, the k-2 f value needs f values in k-3 and k-4 */
/* families,..., until what we need is f value in fmaxk family, which  */
/* is already stored in some node. For any k, there are lots of f       */
/* values to be evaluated. We are doing 16K of them at a time, one on  */
/* each node.                                                           */
/*************************************************************************/
/* PE array can wrap around as well as overlap. A virtural pe array is */
```

```
/* used in concept with length 16384*wplvl. PE array can overlap wplvl */
/* times(in the form of array of dimension wplvl).                      */
/************************************************************************/
/* In interval routines OVERFLOW and UNDERFLOW are not implemented.     */
/* Determinant of sigma inverse is not computed, this part should       */
/* divide the result.                                                   */
/************************************************************************/

#include "/home/ouhong/interval/mytype.h"
#include "/usr/include/mips/fpu.h"
#include <stdio.h>
#include <mpl.h>
#include <math.h>
#define maxn 10
#define maxintv 10
#define pesize 16384

/* Warp around the pe array 50 times. Cannot do 65 times like in */
/* mulnor_wrap.m, because in that case we used all the memory,   */
/* but now we have to leave some to the stack for recursive call */
#define wplvl 50

/* Virtural pe array with size 16384*50. This array can warp, but */
/* cannot overlap.                                                */
#define wsize 819200 /* 16384*50 */
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
extern plural intvl gexp(plural intvl);
extern plural intvl gln(plural intvl);
visible extern feupdown();
visible extern int ffn,ffm[maxn+1];
visible extern double fa[maxn+1][maxintv+1];
visible extern struct {double l,r;} ffc[maxn+1][maxintv+1],
                  ffh[maxn+1][maxintv+1],ffx[maxn+1][maxintv+1];
           /* make some variables global, because they are used */
           /* recursively in getf()                             */
plural intvl getf();
plural int maxk,n,le,ble,donenum;
plural intvl t[maxn+1][maxn+1],c[2][maxn+1],x[2][maxn+1],fr[50][2],
            fe[50][2];
```

```
main()
{ int fn,fmaxk,fi,fj,fm[maxn+1],tm,si[maxn+1];
  double a[maxn+1][maxintv+1],b[maxn+1],ft[maxn+1][maxn+1];
  struct {double l,r;} det,fc[maxn+1][maxintv+1],fh[maxn+1][maxintv+1],
       fx[maxn+1][maxintv+1],mc[2][maxn+1],mh[2][maxn+1],mx[2][maxn+1];
  plural int m,i,j,k,kl,wl,wlt,wlbg,wlnd,lepe,blepe,head,tail,headpe,
             tailpe,thisk,act,acthead,acttail,actpe,wrplvl,id[50],
             kx[maxn+1],ki[50][maxn+1],pid,q,kp[50][maxn+1],l[maxn+1],
             level,pace,out,r=1,e=0;
  plural int p[50],item[maxn+1],dist,node,psum[maxn+1],nodepe,nodepewl,
             ti,twl;
  plural double num,den;
  plural intvl gpartr[2],gparte[2],pfr[2],pfe[2],tmp;

/* num and den are changed to double, but they are used as int. */
/* Using int will sometimes cause overflow.                     */
  plural intvl k0,h[2][maxn+1],neghlf,ptr[2],pte[2],tmp1,tmp2,tmp3r,
     tmp3e,tmprule,tmperr,two,rule,result,sfr[2],sfe[2],ttmprule,ttmperr;

  printf("Input n(between 1 and 10) : ");
  scanf("%d",&fn);

/* fmaxk is the maximum k such that the total # of pe nodes needed */
/* for k,k-1,k-2(for this n) fit in the 128x128 pe array.          */
  switch (fn)
   { case 1  : fmaxk=100000; break;
     case 2  : fmaxk=100000; break;
     case 3  : fmaxk=700;    break;
     case 4  : fmaxk=116;    break;
     case 5  : fmaxk=49;     break;
     case 6  : fmaxk=29;     break;
     case 7  : fmaxk=21;     break;
     case 8  : fmaxk=17;     break;
     case 9  : fmaxk=14;     break;
     case 10 : fmaxk=12;     break;
   }
  printf("Input a1 b1 a2 b2 ... etc.\n");

/* the total # of subregions is restricted by the limited pe memory */

  printf("(with width on each dimention at most 1.9)\n");
  for (fi=1;fi<=fn;fi++) scanf("%lf %lf",&a[fi][0],&b[fi]);
```

```
  printf("Input upper triangular elements of sigma inverse (t11 t12
                                                  t13 ... t22 t23 ...)\n");
  for (fi=1;fi<=fn;fi++)
   for (fj=fi;fj<=fn;fj++)
    { scanf("%lf",&ft[fi][fj]);
      ft[fj][fi]=ft[fi][fj];
    }

/* cut each dimention to make each part with length <= 1.9  */
/* By this we make sure each iteration we get tighter       */
/* inclusion, instead of first wider ones then tighter ones */
  for (fi=1;fi<=maxn;fi++) fm[fi]=1;
  for (fi=1;fi<=fn;fi++)
   { while (b[fi]-a[fi][fm[fi]-1]>1.9)
       { a[fi][fm[fi]]=a[fi][fm[fi]-1]+1.9;
         fm[fi]++;
       }
     a[fi][fm[fi]]=b[fi];
   }

/* get for each dimention the c's, h's and x's. Do this on Front End */
/* because it has rounding mode control.                             */
/*  fe_round_down(); */
/*******************************************/
/*for (fi=1;fi<=fn;fi++)                   */
/* for (fj=1;fj<=fm[fi];fj++)              */
/*  { fc[fi][fj].l=(a[fi][fj-1]+a[fi][fj])/2.0;*/
/*    fh[fi][fj].l=(a[fi][fj]-a[fi][fj-1])/2.0;*/
/*    fx[fi][fj].l=a[fi][fj-1];            */
/*  }                                      */
/*******************************************/
/*  fe_round_up(); */
/*******************************************/
/*for (fi=1;fi<=fn;fi++)                   */
/* for (fj=1;fj<=fm[fi];fj++)              */
/*  { fc[fi][fj].r=(a[fi][fj-1]+a[fi][fj])/2.0;*/
/*    fh[fi][fj].r=(a[fi][fj]-a[fi][fj-1])/2.0;*/
/*    fx[fi][fj].r=a[fi][fj];              */
/*  }                                      */
/*******************************************/
  copyOut(a,fa,8*(maxn+1)*(maxintv+1));
  copyOut(&fn,&ffn,4);
```

```
  copyOut(fm,ffm,4*(maxn+1));
  callRequest(feupdown,0);
  copyIn(ffc,fc,2*8*(maxn+1)*(maxintv+1));
  copyIn(ffh,fh,2*8*(maxn+1)*(maxintv+1));
  copyIn(ffx,fx,2*8*(maxn+1)*(maxintv+1));

/* get the total # of subregions (tm), and c's,      */
/* h's and x's on each dimention for each subregion. */
  tm=0;
  for (si[10]=1;si[10]<=fm[10];si[10]++)
   for (si[9]=1;si[9]<=fm[9];si[9]++)
    for (si[8]=1;si[8]<=fm[8];si[8]++)
     for (si[7]=1;si[7]<=fm[7];si[7]++)
      for (si[6]=1;si[6]<=fm[6];si[6]++)
       for (si[5]=1;si[5]<=fm[5];si[5]++)
        for (si[4]=1;si[4]<=fm[4];si[4]++)
         for (si[3]=1;si[3]<=fm[3];si[3]++)
          for (si[2]=1;si[2]<=fm[2];si[2]++)
           for (si[1]=1;si[1]<=fm[1];si[1]++)
            { tm++;
              for (fi=1;fi<=fn;fi++)
               { mc[tm][fi].l=fc[fi][si[fi]].l;
                 mc[tm][fi].r=fc[fi][si[fi]].r;
                 mh[tm][fi].l=fh[fi][si[fi]].l;
                 mh[tm][fi].r=fh[fi][si[fi]].r;
                 mx[tm][fi].l=fx[fi][si[fi]].l;
                 mx[tm][fi].r=fx[fi][si[fi]].r;
               }
            }

/* Copy the main parameters onto pe array */
  m=tm;
  n=fn;
  maxk=fmaxk;
  for (i=1;i<=m;i++)
   for (j=1;j<=n;j++)
     { c[i][j].l.unf=mc[i][j].l;
       c[i][j].r.unf=mc[i][j].r;
       h[i][j].l.unf=mh[i][j].l;
       h[i][j].r.unf=mh[i][j].r;
       x[i][j].l.unf=mx[i][j].l;
       x[i][j].r.unf=mx[i][j].r;
```

```
    }
  for (i=1;i<=n;i++)
   for (j=i;j<=n;j++)
     t[i][j].l.unf=t[i][j].r.unf=t[j][i].l.unf=t[j][i].r.unf=ft[i][j];
  neghlf.l.unf=neghlf.r.unf= -0.5;

/* get k0, WITHOUT sigma inverse part */
  tmp1.l.unf=tmp1.r.unf=fn;
  tmp2.l.unf=0.3989422804014326;
  tmp2.r.unf=0.3989422804014327;
  k0=gexp(gmul(tmp1,gln(tmp2)));

/* Calculate for k=0 on pe #0, rule is the total rule part without */
/* k0*2**n, fr[0][i] is f(c1,c2,...) and fe[0][i] is f(x1,x2,...)  */
/* for subregion i at 0 position of the f array.                   */
  if (iproc==0)
   { rule.l.unf=rule.r.unf=0.0;
     for (i=1;i<=m;i++)
       { fr[0][i].l.unf=fr[0][i].r.unf=fe[0][i].l.unf=fe[0][i].r.unf=0.0;
         for (j=1;j<=n;j++)
          for (k=1;k<=n;k++)
            { fr[0][i]=gadd(fr[0][i],gmul(t[j][k],gmul(c[i][j],c[i][k])));
              fe[0][i]=gadd(fe[0][i],gmul(t[j][k],gmul(x[i][j],x[i][k])));
            }
         fr[0][i]=gexp(gmul(neghlf,fr[0][i]));
         fe[0][i]=gexp(gmul(neghlf,fe[0][i]));
         tmp.l.unf=fr[0][i].l.unf;
         tmp.r.unf=fr[0][i].r.unf;
         for (j=1;j<=n;j++) tmp=gmul(tmp,h[i][j]);
         rule=gadd(rule,tmp);
       }

/* Each node can have wplvl # of f values. They can wrap but cannot */
/* overlap. wlbg and wlnd indicate the begin and end in the f array */
/* for the next round of computation(ie. for next k). Within the    */
/* round, they indicate the current to-be-computed f values.        */
     wlbg=wlnd=1;
   }

/* le means the ending node for k-1 group in the virtural array, */
/* ble is the ending node for k-2 group in the virtural array.   */
/* lepe and blepe are the pe node # for le and ble where they    */
```

```
/* physically reside. donenum is the total # of f values that    */
/* have been computed so far.                                    */
  donenum=0;
  le=0; lepe=0;
  ble= -1; blepe= -1;
  if (iproc!=0) wlbg=wlnd=0;

/* Now the main part. act is used to indicate which node should be */
/* active. act=1 means active, otherwise not active. head and tail */
/* indicate the head and tail position for this k group in the     */
/* virtural array. headpe and tailpe are their physical residence  */
/* pe node #. thisk is the # of f values for this k group. head    */
/* node has id # 1, tail node has id # thisk, so on. On each pe,   */
/* start from f array position wlbg, end with position wlnd.       */
  for (k=1;k<=2*maxk;k++)
   { if (k>maxk)

/* According to k>maxk or not, we employ different strategy. */
/* If k<=maxk, the program is exactly the same as            */
/* mulnor_wrap.m. If k>maxk, use getf() recursively.         */
      { all {two.l.unf=two.r.unf=2.0;}
        headpe=0;
        num=den=1.0;
        for (i=1;i<=n-1;i++)
          { num=num*(n+k-i);
            den=den*i;
          }
        thisk=num/den;
        id[0]=1+iproc;
        tmprule.l.unf=tmprule.r.unf=tmperr.l.unf=tmperr.r.unf=0.0;
        while (thisk>0)
         { act=0;
           if (thisk>pesize)
            { actpe=pesize;
              acthead=0;
              acttail=pesize-1;
              all {act=1;}
            }
           else
            { actpe=thisk;
              acthead=0;
              acttail=thisk-1;
```

```
            if (iproc<=thisk-1) act=1;
          }
         if (act==1)
          { for (i=1;i<=m;i++)
              sfr[i].l.unf=sfr[i].r.unf=sfe[i].l.unf=sfe[i].r.unf=0.0;
            pid=0;
            out=0;
            for (kx[1]=n>1?0:k;kx[1]<=k && out==0;kx[1]++)
             for (kx[2]=n>2?0:k-kx[1];kx[2]<=k-kx[1] && out==0;kx[2]++)
              for (kx[3]=n>3?0:k-kx[1]-kx[2];kx[3]<=k-kx[1]-kx[2] &&
                                                    out==0;kx[3]++)
               for (kx[4]=n>4?0:k-kx[1]-kx[2]-kx[3];
                         kx[4]<=k-kx[1]-kx[2]-kx[3] && out==0;kx[4]++)
                for (kx[5]=n>5?0:k-kx[1]-kx[2]-kx[3]-kx[4];
                    kx[5]<=k-kx[1]-kx[2]-kx[3]-kx[4] && out==0;kx[5]++)
                 for (kx[6]=n>6?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5];
                      kx[6]<=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5] && out==0;
                                                          kx[6]++)
                  for (kx[7]=n>7?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                        kx[6];kx[7]<=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                                              kx[6] && out==0;kx[7]++)
                   for (kx[8]=n>8?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                         kx[6]-kx[7];kx[8]<=k-kx[1]-kx[2]-kx[3]-kx[4]-
                                    kx[5]-kx[6]-kx[7] && out==0;kx[8]++)
                    for (kx[9]=n>9?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                          kx[6]-kx[7]-kx[8];kx[9]<=k-kx[1]-kx[2]-kx[3]-
                        kx[4]-kx[5]-kx[6]-kx[7]-kx[8] && out==0;kx[9]++)
                     { pid++;
                       if (pid==id[0])
                         { kx[10]=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                                              kx[6]-kx[7]-kx[8]-kx[9];
                           for (i=1;i<=n;i++) ki[0][i]=kx[i];
                           out=1;
                         }
                     }
            for (i=1;i<=m;i++)
/* Here are the two function calls to get f value for ith subregion with */
/* sub ki[0][1],ki[0][2],..., in k family, for the rule(r) or error(e)   */
              { sfr[i]=gadd(sfr[i],getf(ki,k,i,r));
                sfe[i]=gadd(sfe[i],getf(ki,k,i,e));
              }
            for (i=1;i<=m;i++)
```

```
{ ptr[i].l.unf=ptr[i].r.unf=pte[i].l.unf=pte[i].r.unf=0.0;
  tmp3r.l.unf=sfr[i].l.unf;
  tmp3r.r.unf=sfr[i].r.unf;
  tmp3e.l.unf=sfe[i].l.unf;
  tmp3e.r.unf=sfe[i].r.unf;
  q=0;
  j=1;
  while (j<=n)
    { if (ki[0][j]!=(ki[0][j]/2)*2) q=1;
      j++;
    }
  for (j=1;j<=n;j++)
    { tmp1.l.unf=tmp1.r.unf=ki[0][j]+1.0;
      if (k==(k/2)*2 && q==0) tmp3r=gdiv(gmul(tmp3r,
                    gexp(gmul(tmp1,gln(h[i][j]))),tmp1);
      if (ki[0][j]==(ki[0][j]/2)*2)
         tmp3e=gmul(gmul(tmp3e,two),gexp(gmul(tmp1,
                                          gln(h[i][j]))));
      else
        { tmp2=gmul(tmp3e,gexp(gmul(tmp1,gln(h[i][j]))));
          tmp3e=gsub(tmp2,tmp2);
        }
    }
  for (j=1;j<=n;j++)
    { tmp1.l.unf=tmp1.r.unf=ki[0][j]+1.0;
      tmp3e=gdiv(tmp3e,tmp1);
    }
  if (k==(k/2)*2 && q==0) ptr[i]=gadd(ptr[i],tmp3r);
  pte[i]=gadd(pte[i],tmp3e);
}
ttmprule.l.unf=ttmprule.r.unf=ttmperr.l.unf=
                               ttmperr.r.unf=0.0;
for (i=1;i<=m;i++) ttmperr=gadd(ttmperr,pte[i]);
if (k==(k/2)*2)
  for (i=1;i<=m;i++) ttmprule=gadd(ttmprule,ptr[i]);
level=0;
j=1;
while (j<actpe)
  { j=j*2;
    level++;
  }
for (i=0;i<=level-1;i++)
```

```
    { if (i==0) pace=1;
      else pace=pace*2;
      j=iproc+pace;
      if (j>acttail) tmp.l.unf=tmp.r.unf=0.0;
      else
        { tmp.l.unf=router[j].ttmperr.l.unf;
          tmp.r.unf=router[j].ttmperr.r.unf;
        }
      ttmperr=gadd(ttmperr,tmp);
    }
  tmperr=gadd(tmperr,ttmperr);
  if (iproc==0 && thisk<=pesize)
    { rule.l.unf=result.l.unf=router[(blepe+1)<pesize?
                                  blepe+1:0].rule.l.unf;
      rule.r.unf=result.r.unf=router[(blepe+1)<pesize?
                                  blepe+1:0].rule.r.unf;
      all { blepe= -1;}
      for (i=1;i<=n;i++) result=gmul(two,result);
      result=gmul(gadd(result,tmperr),k0);
      p_printf("Result : %30.27f -- %30.27f\n",result.l.unf,
                                                result.r.unf);
    }
  if (k==(k/2)*2)
    { for (i=0;i<=level-1;i++)
        { if (i==0) pace=1;
          else pace=pace*2;
          j=iproc+pace;
          if (j>acttail) tmp.l.unf=tmp.r.unf=0.0;
          else
            { tmp.l.unf=router[j].ttmprule.l.unf;
              tmp.r.unf=router[j].ttmprule.r.unf;
            }
          ttmprule=gadd(ttmprule,tmp);
        }
      tmprule=gadd(tmprule,ttmprule);
      if (iproc==0 && thisk<=pesize) rule=gadd(rule,tmprule);
    }
  all
    { thisk -= pesize;
      id[0] += pesize;
    }
}
```

```
    }
 }
else {
act=0;
head=(le+1)<wsize?le+1:0;
headpe=head-(head/pesize)*pesize;
num=den=1.0;
for (i=1;i<=n-1;i++)
  { num=num*(n+k-i);
    den=den*i;
  }
thisk=num/den;
tail=(le+thisk)<wsize?le+thisk:le+thisk-wsize;
tailpe=tail-(tail/pesize)*pesize;
if (thisk>=pesize)
  { id[wlbg]=iproc>lepe?iproc-lepe:pesize+(iproc-lepe);
    j=2;
    while (thisk>=j*pesize)
     { wlnd=(wlnd+1)<wplvl?wlnd+1:0;
       id[wlnd]=id[(wlnd-1)>=0?wlnd-1:wplvl-1]+pesize;
       j++;
     }
    if (headpe<=tailpe) {if (iproc>=headpe && iproc<=tailpe) act=1;}
    if (headpe>tailpe+1) {if ((iproc>=headpe && iproc<=pesize-1) ||
                              (iproc>=0 && iproc<=tailpe)) act=1;}
    if (act==1)
      { wlnd=(wlnd+1)<wplvl?wlnd+1:0;
        id[wlnd]=id[(wlnd-1)>=0?wlnd-1:wplvl-1]+pesize;
      }
    act=1;
    acthead=headpe;
    acttail=(headpe-1)>=0?headpe-1:pesize-1;
    actpe=pesize;
  }
else
  { if (headpe<tailpe) { if (iproc>=headpe && iproc<=tailpe) act=1; }
    else { if ((iproc>=headpe && iproc<=pesize-1) || (iproc>=0 &&
                                  iproc<=tailpe)) act=1; }
    if (act==1) id[wlbg]=iproc>lepe?iproc-lepe:pesize+(iproc-lepe);
    acthead=headpe;
    acttail=tailpe;
    actpe=thisk;
```

```
      }
    if (act==1)
      { pid=0;
        j=wlbg;

/* The next huge loop is used to find k1,k2,...,kn */
/* (ki[j][1] to ki[j][n]) for each active node for */
/* each fold j.                                    */
        for (kx[1]=n>1?0:k;kx[1]<=k;kx[1]++)
         for (kx[2]=n>2?0:k-kx[1];kx[2]<=k-kx[1];kx[2]++)
          for (kx[3]=n>3?0:k-kx[1]-kx[2];kx[3]<=k-kx[1]-kx[2];kx[3]++)
           for (kx[4]=n>4?0:k-kx[1]-kx[2]-kx[3];kx[4]<=k-kx[1]-kx[2]-
                                                       kx[3];kx[4]++)
            for (kx[5]=n>5?0:k-kx[1]-kx[2]-kx[3]-kx[4];kx[5]<=k-kx[1]-
                                                 kx[2]-kx[3]-kx[4];kx[5]++)
             for (kx[6]=n>6?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5];kx[6]<=
                                k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5];kx[6]++)
              for (kx[7]=n>7?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-kx[6];
                     kx[7]<=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-kx[6];kx[7]++)
               for (kx[8]=n>8?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-kx[6]-
                       kx[7];kx[8]<=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                                                 kx[6]-kx[7];kx[8]++)
                for (kx[9]=n>9?0:k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-
                        kx[6]-kx[7]-kx[8];kx[9]<=k-kx[1]-kx[2]-kx[3]-
                                kx[4]-kx[5]-kx[6]-kx[7]-kx[8];kx[9]++)
                 { pid++;
                   if (pid==id[j])
                     { kx[10]=k-kx[1]-kx[2]-kx[3]-kx[4]-kx[5]-kx[6]-
                                                       kx[7]-kx[8]-kx[9];
                       for (i=1;i<=n;i++) ki[j][i]=kx[i];
                       j=(j+1)<wplvl?j+1:0;
                     }
                 }

/* p is the first position that ki[j][p]!=0 for fold j. */
        for (i=wlbg;i<=(wlnd>=wlbg?wlnd:wlnd+wplvl);i++)
          { wlt=i<wplvl?i:i-wplvl;
            p[wlt]=1;
            while (ki[wlt][p[wlt]]==0) p[wlt]++;
          }

/* Get k prime (kp[j][1] to kp[j][n]) for fold j. */
```

```
          for (j=wlbg;j<=(wlnd>=wlbg?wlnd:wlnd+wplvl);j++)
            { wlt=j<wplvl?j:j-wplvl;
              for (i=1;i<=n;i++) kp[wlt][i]=ki[wlt][i];
              kp[wlt][p[wlt]]=ki[wlt][p[wlt]]-1;
            }

/* Have to repeat the following procedure to get f values for position   */
/* wlbg through wlnd of the f array.                                     */
/* ptr[j][i] and pte[j][i] is the f value to be computed on this pe node */
/* (for subregion i and fold j)                                          */
          for (wl=wlbg;wl<=(wlnd>=wlbg?wlnd:wlnd+wplvl);wl++)
           { wlt=wl<wplvl?wl:wl-wplvl;
             for (i=1;i<=m;i++)
                ptr[i].l.unf=ptr[i].r.unf=pte[i].l.unf=pte[i].r.unf=0.0;
             for (i=1;i<=n;i++) l[i]=kp[wlt][i];

/* psum[i] is partial sum of k prime from i to n.           */
/* item's are used to identify the node for previous f value */
/* with (kp[1],..,kp[n]) in k-1 goup. They can also be used  */
/* to find nodes for k-2 group.                             */
             psum[n]=kp[wlt][n];
             for (i=n-1;i>=2;i--) psum[i]=psum[i+1]+kp[wlt][i];
             for (i=1;i<=n-1;i++)
               { num=den=1;
                 for (j=0;j<=n-(i+1);j++)
                   { num=num*(psum[i+1]+j);
                     den=den*(j+1);
                   }
                 item[i]=num/den;
               }

/* There are at most n+1 terms to form the f value on this pe node */
             for (q=0;q<=n;q++)

/* Different cases have different g values. l[i] record the subscript */
/* for the currently needed previous f value.                         */
               { if (q==0)
                   for (j=1;j<=m;j++)
                     { gpartr[j].l.unf=gpartr[j].r.unf=gparte[j].l.unf
                                                      =gparte[j].r.unf=0.0;
                       for (i=1;i<=n;i++)
                         { gpartr[j]=gsub(gpartr[j],gmul(t[i][p[wlt]],
```

```
                                                    c[j][i]));
                    gparte[j]=gsub(gparte[j],gmul(t[i][p[wlt]],
                                                    x[j][i]));
                  }
              }
            else
              { if (l[q]==0) continue;
                else
                  { for (j=1;j<=m;j++)
                      { gpartr[j].l.unf=gparte[j].l.unf=
                                        -t[q][p[wlt]].r.unf;
                        gpartr[j].r.unf=gparte[j].r.unf=
                                        -t[q][p[wlt]].l.unf;
                      }
                    l[q]--;
                  }
              }

/* kl is the group # for the currently needed previous f value. */
            kl=0;
            for (i=1;i<=n;i++) kl+=l[i];

/* dist is the distance between the currently needed f value */
/* and the last f value in their group.                      */
            dist=0;
            if (kl==k-1)
              { for (i=1;i<=n-1;i++) dist=dist+item[i];
                node=(le-dist)>=0?le-dist:wsize+(le-dist);
              }
            else
              { for (i=1;i<=n-1;i++)
                  { if (i<=q-1) dist=dist+item[i]*(psum[i+1]-1)
                                /(psum[i+1]+n-(i+1));
                    else dist=dist+item[i];
                  }
                node=(ble-dist)>=0?ble-dist:wsize+(ble-dist);
              }

/* Use donenum(the total # of f values from step 0 to step k, */
/* or from 0 to le) to get the residing pe # for the needed   */
/* node in the virtural array. nodepe is the pe #, nodepewl   */
/* is the index in the f array(or fold #).                    */
```

```
                i=donenum-(le>=node?le-node:wsize+(le-node));
                j=i/pesize;
                nodepe=i-j*pesize;
                nodepewl=j<wplvl?j:j-(j/wplvl)*wplvl;

/* Fetch the needed f value, form ptr[i] and pte[i].   */
/* pfr is previous fr value, pfe is previous fe value. */
                for (i=1;i<=m;i++)
                  { router[nodepe].ti=i;
                    router[nodepe].twl=nodepewl;
                    pfr[i].l.unf=router[nodepe].fr[twl][ti].l.unf;
                    pfr[i].r.unf=router[nodepe].fr[twl][ti].r.unf;
                    pfe[i].l.unf=router[nodepe].fe[twl][ti].l.unf;
                    pfe[i].r.unf=router[nodepe].fe[twl][ti].r.unf;
                    ptr[i]=gadd(ptr[i],gmul(pfr[i],gpartr[i]));
                    pte[i]=gadd(pte[i],gmul(pfe[i],gparte[i]));
                  }
                if (q!=0) l[q]++;
              }

/* They need to divide something to be what they should be. */
            tmp.l.unf=tmp.r.unf=ki[wlt][p[wlt]];
            for (i=1;i<=m;i++)
              { ptr[i]=gdiv(ptr[i],tmp);
                pte[i]=gdiv(pte[i],tmp);
/* Form  the f value for this node (fr[i] and fe[i]) (for subregion i). */
                fr[wlt][i].l.unf=ptr[i].l.unf;
                fr[wlt][i].r.unf=ptr[i].r.unf;
                fe[wlt][i].l.unf=pte[i].l.unf;
                fe[wlt][i].r.unf=pte[i].r.unf;
              }
          }

/* On each active node, for each subregion, form the rule */
/* and error part to be added. Rule part is formed only   */
/* when k is even and all ki's are even.                  */
         two.l.unf=two.r.unf=2.0;
         for (i=1;i<=m;i++)
          { ptr[i].l.unf=ptr[i].r.unf=pte[i].l.unf=pte[i].r.unf=0.0;
            for (wl=wlbg;wl<=(wlnd>=wlbg?wlnd:wlnd+wplvl);wl++)
             { wlt=wl<wplvl?wl:wl-wplvl;
               tmp3r.l.unf=fr[wlt][i].l.unf;
```

```
            tmp3r.r.unf=fr[wlt][i].r.unf;
            tmp3e.l.unf=fe[wlt][i].l.unf;
            tmp3e.r.unf=fe[wlt][i].r.unf;
            q=0;
            j=1;
            while (j<=n)
              { if (ki[wlt][j]!=(ki[wlt][j]/2)*2) q=1;
                j++;
              }
            for (j=1;j<=n;j++)
              { tmp1.l.unf=tmp1.r.unf=ki[wlt][j]+1.0;
                if (k==(k/2)*2 && q==0) tmp3r=gdiv(gmul(tmp3r,
                           gexp(gmul(tmp1,gln(h[i][j])))),tmp1);
                if (ki[wlt][j]==(ki[wlt][j]/2)*2)
                   tmp3e=gmul(gmul(tmp3e,two),gexp(gmul(tmp1,
                                              gln(h[i][j]))));
                else
                  { tmp2=gmul(tmp3e,gexp(gmul(tmp1,gln(h[i][j]))));
                    tmp3e=gsub(tmp2,tmp2);
                  }
              }
            for (j=1;j<=n;j++)
              { tmp1.l.unf=tmp1.r.unf=ki[wlt][j]+1.0;
                tmp3e=gdiv(tmp3e,tmp1);
              }
            if (k==(k/2)*2 && q==0) ptr[i]=gadd(ptr[i],tmp3r);
            pte[i]=gadd(pte[i],tmp3e);
          }
        }

/* tmperr is the summation of pte[i] over all subregions. */
/* tmprule is similar but is formed only when k is even.  */
/* Not checking all the ki's are even, since the values   */
/* are set to 0(see above) if this is the case.           */
        tmprule.l.unf=tmprule.r.unf=tmperr.l.unf=tmperr.r.unf=0.0;
        for (i=1;i<=m;i++) tmperr=gadd(tmperr,pte[i]);
        if (k==(k/2)*2)
          for (i=1;i<=m;i++) tmprule=gadd(tmprule,ptr[i]);

/* Add over all active nodes the tmperr value to */
/* form the total tmperr value store it on the   */
/* head node. Algorithm is binary tree.          */
```

```
        level=0;
        j=1;
        while (j<actpe)
          { j=j*2;
            level++;
          }
        for (i=0;i<=level-1;i++)
          { if (i==0) pace=1;
            else pace=pace*2;
            j=iproc+pace;
            if ((acthead<acttail && j>acttail) || (acthead>acttail &&
                            ((iproc>=acthead && j>acttail+pesize) ||
                                    (iproc<=acttail && j>acttail))))
              tmp.l.unf=tmp.r.unf=0.0;
            else
              { j=j<pesize?j:j-pesize;
                tmp.l.unf=router[j].tmperr.l.unf;
                tmp.r.unf=router[j].tmperr.r.unf;
              }
            tmperr=gadd(tmperr,tmp);
          }

/* The head node fetches the previously accumulated rule variable, */
/* add the total tmperr to produce the current result.             */
        if (iproc==headpe)
          { rule.l.unf=result.l.unf=router[(blepe+1)<pesize?
                                           blepe+1:0].rule.l.unf;
            rule.r.unf=result.r.unf=router[(blepe+1)<pesize?
                                           blepe+1:0].rule.r.unf;
            for (i=1;i<=n;i++) result=gmul(two,result);
            result=gmul(gadd(result,tmperr),k0);
            p_printf("Result : %30.27f -- %30.27f\n",
                                       result.l.unf,result.r.unf);
          }

/* If k is even, rule variable needs to be updated on the head node.  */
/* Algorithm similar as error part. But only add those nodes with     */
/* all ki[i] even, hense if this is not the case, set the value to 0. */
        if (k==(k/2)*2)
          { for (i=0;i<=level-1;i++)
              { if (i==0) pace=1;
                else pace=pace*2;
```

```
                j=iproc+pace;
                if ((acthead<acttail && j>acttail) || (acthead>acttail
                     && ((iproc>=acthead && j>acttail+pesize) ||
                             (iproc<=acttail && j>acttail))))
                  tmp.l.unf=tmp.r.unf=0.0;
                else
                  { j=j<pesize?j:j-pesize;
                    tmp.l.unf=router[j].tmprule.l.unf;
                    tmp.r.unf=router[j].tmprule.r.unf;
                  }
                tmprule=gadd(tmprule,tmp);
              }
            if (iproc==headpe) rule=gadd(rule,tmprule);
          }

/* Update flags */
         wlbg=wlnd=(wlnd+1)<wplvl?wlnd+1:0;
         all
         { ble=le;
           blepe=lepe;
           le=tail;
           lepe=tailpe;
           donenum+=thisk;
         }
       }
   }
   }
}

plural intvl getf(ki,k,i,ch)
plural int ki[maxn+1],k,i,ch;
{ plural intvl getftmp,tmp,pfr[2],pfe[2],gpartr[2],gparte[2];
  plural int p,kl,kp[maxn+1],l[maxn+1],psum[maxn+1],item[maxn+1],
             dist,node,nodepe,nodepewl,ti,twl,q,j,ii;
  plural double num,den;
  p=1;
  while (ki[p]==0) p++;
  for (j=1;j<=n;j++) kp[j]=ki[j];
  kp[p]=ki[p]-1;
  for (j=1;j<=n;j++) l[j]=kp[j];
  psum[n]=kp[n];
  for (j=n-1;j>=2;j--) psum[j]=psum[j+1]+kp[j];
```

```
for (ii=1;ii<=n-1;ii++)
  { num=den=1;
    for (j=0;j<=n-(ii+1);j++)
     { num=num*(psum[ii+1]+j);
       den=den*(j+1);
     }
    item[ii]=num/den;
  }
getftmp.l.unf=getftmp.r.unf=0.0;
for (q=0;q<=n;q++)
  { if (q==0)
      { gpartr[i].l.unf=gpartr[i].r.unf=gparte[i].l.unf
                                        =gparte[i].r.unf=0.0;
        for (j=1;j<=n;j++)
         { gpartr[i]=gsub(gpartr[i],gmul(t[j][p],c[i][j]));
           gparte[i]=gsub(gparte[i],gmul(t[j][p],x[i][j]));
         }
      }
    else
      { if (l[q]==0) continue;
        else
          { gpartr[i].l.unf=gparte[i].l.unf= -t[q][p].r.unf;
            gpartr[i].r.unf=gparte[i].r.unf= -t[q][p].l.unf;
            l[q]--;
          }
      }
    kl=0;
    for (j=1;j<=n;j++) kl+=l[j];
    if (kl<=maxk)
      { dist=0;
        if (kl==k-1)
          { for (j=1;j<=n-1;j++) dist=dist+item[j];
            node=(le-dist)>=0?le-dist:wsize+(le-dist);
          }
        else
          { for (j=1;j<=n-1;j++)
              { if (j<=q-1) dist=dist+item[j]*(psum[j+1]-1)/
                                          (psum[j+1]+n-(j+1));
                else dist=dist+item[j];
              }
            if (k==maxk+2) node=(le-dist)>=0?le-dist:wsize+(le-dist);
            else node=(ble-dist)>=0?ble-dist:wsize+(ble-dist);
```

```
            }
          ii=donenum-(le>=node?le-node:wsize+(le-node));
          j=ii/pesize;
          nodepe=ii-j*pesize;
          nodepewl=j<wplvl?j:j-(j/wplvl)*wplvl;
          if (ch==1)
            { router[nodepe].ti=i;
              router[nodepe].twl=nodepewl;
              pfr[0].l.unf=router[nodepe].fr[twl][ti].l.unf;
              pfr[0].r.unf=router[nodepe].fr[twl][ti].r.unf;
              getftmp=gadd(getftmp,gmul(pfr[0],gpartr[i]));
            }
          else
            { router[nodepe].ti=i;
              router[nodepe].twl=nodepewl;
              pfe[0].l.unf=router[nodepe].fe[twl][ti].l.unf;
              pfe[0].r.unf=router[nodepe].fe[twl][ti].r.unf;
              getftmp=gadd(getftmp,gmul(pfe[0],gparte[i]));
            }
        }
      else
        { if (ch==1)
            { pfr[0]=getf(l,kl,i,ch);
              getftmp=gadd(getftmp,gmul(pfr[0],gpartr[i]));
            }
          else
            { pfe[0]=getf(l,kl,i,ch);
              getftmp=gadd(getftmp,gmul(pfe[0],gparte[i]));
            }
        }
      if (q!=0) l[q]++;
    }
  tmp.l.unf=tmp.r.unf=ki[p];
  getftmp=gdiv(getftmp,tmp);
  return getftmp;
}
```

APPENDIX D. MASPAR SOURCE CODES FOR OPTIMIZATION

We include here only the program for Example 2 of Section 5.3.

```
/*************************************************************************/
/*                              nlin15.g.m                               */
/*************************************************************************/
/* For different nonlinear regression problems, only the     */
/* parts between the ==== lines have to be changed. Input    */
/* is required for the dimension of the X matrix, the number */
/* of parameters to be estimated, and the regions in which   */
/* the parameters should be searched. The elements of the X  */
/* matrix can be either put in the program or input by the   */
/* user interactively. Output: after each step the current   */
/* inclusion of the optimal function value is printed,       */
/* together with the # of PE nodes that are not cleared.     */
/* When finished, the program outputs all the regions and    */
/* the function inclusions on those regions. Then it tries   */
/* to group the regions dimension by dimension and outputs   */
/* the regions after grouping, and the function inclusions   */
/* on those regions.                                         */
#include "/home/disk2/ouhong/interval/oparith/mytype.h"
#include <float.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpl.h>
#include <mp_libc.h>
#include <math.h>
#define mpsize 4096
#define mpsizepow 12
#define epsop 0.000001
#define RANDMAX 2147483647
/* groupow is used to group the final regions. If it is big, */
```

```
/* the # of output regions will be small. Try different      */
/* values to see whether or not we have multiple optimal     */
/* points. Very big groupow usually indicates very clearly   */
/* whether we have several global optima.                    */
#define groupow 3
extern int selectOne();
extern plural intvl gadd(plural intvl,plural intvl);
extern plural intvl gsub(plural intvl,plural intvl);
extern plural intvl gmul(plural intvl,plural intvl);
extern plural intvl gdiv(plural intvl,plural intvl);
extern plural intvl grec(plural intvl);
extern plural intvl gsqu(plural intvl);
extern plural intvl gsqt(plural intvl);
intvl *region,**x,*y;
plural intvl *d;
main()
{ extern plural intvl getfun(plural intvl *,int,int,int);
  extern plural int drvok(plural intvl *,int,int,int);
  extern plural int linfun(plural intvl *,double,int,int,int);
  extern plural int active(plural intvl *,int,int,int);
  int n,p,m,i,j,*k,*gnum,start,going,where,*cutnum,aim,goon=1,left;
  double *piece,awidth,oplow= -DBL_MAX,ophigh=DBL_MAX,oplowtp,tmp;
  plural intvl *box,*center,range;
  plural int quot,flag,rk,*group,tflag,dist,dest,pwhere,derivok;
  plural double maxwidth;

  printf("Input # of rows & columns for the X matrix : ");
  scanf("%d %d",&n,&p);
  printf("Input # of variables in the Nonlinear Regression problem : ");
  scanf("%d",&m);
  region=(intvl *)malloc(m*sizeof(intvl));
  k=(int *)malloc(m*sizeof(int));
  gnum=(int *)malloc(m*sizeof(int));
  cutnum=(int *)malloc(m*sizeof(int));
  piece=(double *)malloc(m*sizeof(double));
  box=(plural intvl *)p_malloc(m*sizeof(intvl));
  center=(plural intvl *)p_malloc(m*sizeof(intvl));
  group=(plural int *)p_malloc(m*sizeof(int));
  y=(intvl *)malloc(n*sizeof(intvl));
  x=(intvl **)malloc(n*sizeof(intvl *));
  for (i=0;i<=n-1;i++) x[i]=(intvl *)malloc(p*sizeof(intvl));
  d=(plural intvl *)p_malloc(m*sizeof(plural intvl));
```

```
/*************************************************/
/* The following assignments need to be changed for   */
/* different problems(the part between the === lines).*/
/*=================================================*/
y[0].l=y[0].r=0.14;
y[1].l=y[1].r=0.18;
y[2].l=y[2].r=0.22;
y[3].l=y[3].r=0.25;
y[4].l=y[4].r=0.29;
y[5].l=y[5].r=0.32;
y[6].l=y[6].r=0.35;
y[7].l=y[7].r=0.39;
y[8].l=y[8].r=0.37;
y[9].l=y[9].r=0.58;
y[10].l=y[10].r=0.73;
y[11].l=y[11].r=0.96;
y[12].l=y[12].r=1.34;
y[13].l=y[13].r=2.10;
y[14].l=y[14].r=4.39;
for (i=0;i<=14;i++)
{ x[i][0].l=x[i][0].r=i+1;
  x[i][1].l=x[i][1].r=15-i;
  if(i<=7) x[i][2].l=x[i][2].r=i+1;
  else x[i][2].l=x[i][2].r=15-i;
}
/*=================================================*/
/*  printf("Input the Y and X matrices, like y, x1,x2,...,xn :\n");*/
/*  for (i=0;i<=n-1;i++)*/
/*   { scanf("%lf %lf",&(y[i].l),&(y[i].r));*/
/*     for (j=0;j<=p-1;j++) scanf("%lf %lf",&(x[i][j].l),&(x[i][j].r));*/
/*   }*/
  printf("Input the starting regions for the variables :\n");
  for (i=0;i<=m-1;i++)
    { scanf("%lf %lf",&(region[i].l),&(region[i].r));
      piece[i]=region[i].r-region[i].l;
      cutnum[i]=1;
    }
/* Decide where to cut the 12 cuts to make 4096 small pieces. */
/* We want to make the length on each dimension as close as   */
/* possible to each other.                                    */
  for (i=1;i<=mpsizepow;i++)
    { where=0;
```

```
      awidth=piece[0];
      for (j=1;j<=m-1;j++) if (piece[j]>awidth)
                           { where=j;
                             awidth=piece[j];
                           }
      piece[where]/=2;
      cutnum[where]*=2;
    }
/* Put one of the 4096 pieces on each pe node. */
  quot=iproc;
  for (i=m-1;i>=0;i--)
    { box[i].l=region[i].l+(quot%cutnum[i])*piece[i];
      if (quot%cutnum[i]!=cutnum[i]-1)
        box[i].r=region[i].l+(quot%cutnum[i]+1)*piece[i];
      else box[i].r=region[i].r;
      quot=quot/cutnum[i];
    }
/* Later points chosen to evaluate the function are random. */
  p_srandom(iproc);
  while (goon)
   { flag=0;
     rk=mpsize;
     if (active(box,n,p,m)!=0)
       { for (i=0;i<=m-1;i++) center[i].l=center[i].r=
                   box[i].l+(box[i].r-box[i].l)*(p_random()*1.0/RANDMAX);
/* Update the upper bound of our minimum. */
         if (active(center,n,p,m)==2)
           { range=getfun(center,n,p,m);
             tmp=reduceMind(range.r);
             ophigh=ophigh>tmp?tmp:ophigh;
           }
         range=getfun(box,n,p,m);
         derivok=drvok(box,n,p,m);
         oplowtp= -DBL_MAX;
         if (range.l<ophigh && derivok==1)
           { oplowtp=reduceMind(range.l);
             oplow=oplow<oplowtp?oplowtp:oplow;
           }
/* If result is good enough, or no more free pe nodes, output. */
         if (reduceMaxd(range.r)-reduceMind(range.l)<=epsop
                   || reduceAdd32(range.l<ophigh && derivok==1)==mpsize)
           { if (range.l<ophigh && derivok==1)
```

```
            { flag=1;
              printf("optimal value is between %25.22f and %25.22f\n",
                                                        oplow,ophigh);
              printf("The following region(s) have function values
                      between(at most) : \n          %25.22f --
                   %25.22f\n",reduceMind(range.l),reduceMaxd(range.r));
              while (flag==1)
                { aim=selectOne();
                  for (i=0;i<=m-1;i++) printf("%25.22f  --  %25.22f\n",
                                proc[aim].box[i].l,proc[aim].box[i].r);
                  printf("f value : %25.22f  --  %25.22f\n",
                                  proc[aim].range.l,proc[aim].range.r);
                  printf("------\n");
                  proc[aim].flag=0;
                }
/* Group the regions to form just a few and output. */
              for (i=0;i<=m-1;i++)
                { group[i]= -1;
                  gnum[i]=0;
                  awidth=reduceMaxd(box[i].r-box[i].l);
                  while (group[i] == -1)
                   { if (box[i].l==reduceMind(box[i].l))
                       start=selectOne();
/* If the distance between two intervals is smaller than */
/* (groupow * length of largest interval), they are      */
/* considered in the same group.                         */
/* For the active PE set control of the plural while     */
/* statement refer to the MasPar manual.                 */
                     going=1;
                     while (going==1)
                      { going=0;
                        if (group[i]== -1 && box[i].l<=
                             proc[start].box[i].r+groupow*awidth)
                         { group[i]=gnum[i];
                           if (box[i].r==reduceMaxd(box[i].r))
                             start=selectOne();
                           going=1;
                         }
                      }
                     gnum[i]++;
                   }
                }
```

```
                printf("####################################\n");
                all { flag=0;}
/*********************************************************************/
/* The following # of for loops needs to be changed according to the    */
/* actual m. The conditions in the if statement also has to be changed  */
/* according to different problems. (The part between the === lines)    */
/*=====================================================================*/
                for (k[0]=0;k[0]<=gnum[0]-1;k[0]++)
                for (k[1]=0;k[1]<=gnum[1]-1;k[1]++)
                for (k[2]=0;k[2]<=gnum[2]-1;k[2]++)
                  if (group[0]==k[0]&&group[1]==k[1]&&group[2]==k[2])
/*=====================================================================*/
                    { aim=selectOne();
                      proc[aim].flag=1;
                      for (i=0;i<=m-1;i++)
                       { proc[aim].box[i].l=reduceMind(box[i].l);
                         proc[aim].box[i].r=reduceMaxd(box[i].r);
                       }
                    }
                if (flag==1) range=getfun(box,n,p,m);
                while (flag==1)
                  { aim=selectOne();
                    for (i=0;i<=m-1;i++) printf("%25.22f  --  %25.22f\n",
                                  proc[aim].box[i].l,proc[aim].box[i].r);
                    printf("f value : %25.22f  --  %25.22f\n",
                                  proc[aim].range.l,proc[aim].range.r);
                    printf("------\n");
                    proc[aim].flag=0;
                  }
              }
            goon=0;
          }
        else
/* Remaining nodes have flag=1. Cleared nodes have flag=0;  */
/* Split the remaining regions to the cleared nodes. Idea   */
/* is : If the # of remained nodes is fewer than cleared    */
/* nodes, then each remaining node look at its next node.   */
/* If it's cleared, split onto it and mark both. If it's    */
/* not, look at its 2nd neighbor. If not again, look at 3rd */
/* neighbor, and so on, until there is no more cleared      */
/* nodes. If the # of remained nodes is larger than cleared */
/* nodes, we want to split those worst nodes and keep the   */
```

```
/* other nodes unchanged(this is done by setting flag=2).   */
/* Worse means if the upper bound of the function value on  */
/* this node is bigger. When this is done, every node will  */
/* have something to compute.                               */
/* When splitting, the strategy is in Hansen's book 9.13.   */
/* Compared with just splitting the biggest dimension, the  */
/* performance is improved dramatically.                    */
          { if (range.l<ophigh && derivok==1) flag=1;
            if (flag==1) flag=linfun(box,ophigh,n,p,m);
/* linfun is the method described in Hansen 9.5(the second  */
/* refinement). It express f in Taylor expansion and solve  */
/* a linear inequality to throw away part of the current    */
/* region. Its use improved the performance greatly.        */
            left=reduceAdd32(flag);
            if (left>mpsize/2)
              if (flag==1)
                { rk=rankd(-range.r);
                  if (rk>=mpsize-left) flag=2;
                }
            printf("f value is between %25.22f and %25.22f\n",
                                              oplow,ophigh);
            printf("# of nodes left=%d\n",left);
            all { while (reduceMin32(flag)==0)
                   if (flag==1)
                     { tflag=1;
                       dist=1;
                       while (tflag==1 && dist<mpsize)
                         { if (router[(iproc+dist)%mpsize].flag==0)
                             { dest=(iproc+dist)%mpsize;
                               pwhere=0;
                               if (d[0].l!= -DBL_MAX && d[0].l!=DBL_MAX
                                  && d[0].r!= -DBL_MAX && d[0].r!=DBL_MAX)
                                { maxwidth=(box[0].r-box[0].l)*
                                                    (d[0].r-d[0].l);
                                  for (i=1;i<=m-1;i++)
                                     if ((box[i].r-box[i].l)*(d[i].r-
                                                    d[i].l)>maxwidth)
                                       { pwhere=i;
                                         maxwidth=(box[i].r-box[i].l)*
                                                    (d[i].r-d[i].l);
                                       }
                                }
```

```
                                else
                                 { maxwidth=box[0].r-box[0].l;
                                   for (i=1;i<=m-1;i++)
                                      if (box[i].r-box[i].l>maxwidth)
                                        { pwhere=i;
                                          maxwidth=box[i].r-box[i].l;
                                        }
                                 }
                                for (i=0;i<=m-1;i++)
                                  { if (i==pwhere)
                                      { router[dest].box[i].l=
                                                (box[i].l+box[i].r)/2;
                                        router[dest].box[i].r=box[i].r;
                                      }
                                     else
                                      { router[dest].box[i].l=box[i].l;
                                        router[dest].box[i].r=box[i].r;
                                      }
                                  }
                                box[pwhere].r=(box[pwhere].l+
                                                    box[pwhere].r)/2;
                                tflag=0;
                                router[dest].flag=1;
                              }
                            else dist++;
                          }
                      }
                  }
              }
          }
      }
}

plural intvl getfun(theta,n,p,m)
plural intvl *theta;
int n,p,m;
{ int i;
  plural intvl fun;
  fun.l=fun.r=0;
  for (i=0;i<=n-1;i++)
/************************************************/
/* The expression of the sum of squares has to be  */
```

```
/* changed according to different problems.        */
/*================================================*/
    fun=gadd(fun,gsqu(gsub(y[i],gadd(theta[0],gdiv(x[i][0],
            gadd(gmul(theta[1],x[i][1]),gmul(theta[2],x[i][2])))))));
/*================================================*/
  return fun;
}

plural int drvok(theta,n,p,m)
plural intvl *theta;
int n,p,m;
/* Use the derivative info to help us throw away some nodes. */
{ plural int tag=1;
  plural intvl neg,tmp1,tmp2;
  int i;
  neg.l=neg.r= -1;
  for (i=0;i<=m-1;i++) d[i].l=d[i].r=0;
  for (i=0;i<=n-1;i++)
/************************************************/
/* The derivatives have to be re-programmed for    */
/* different problems.(The part between === lines) */
/*================================================*/
   { tmp1=gsub(y[i],gadd(theta[0],gdiv(x[i][0],gadd(gmul(theta[1],
              x[i][1]),gmul(theta[2],x[i][2])))));
     tmp2=gsqu(gadd(gmul(theta[1],x[i][1]),gmul(theta[2],x[i][2])));
     d[0]=gadd(d[0],gmul(tmp1,neg));
     d[1]=gadd(d[1],gdiv(gmul(gmul(tmp1,x[i][0]),x[i][1]),tmp2));
     d[2]=gadd(d[2],gdiv(gmul(gmul(tmp1,x[i][0]),x[i][2]),tmp2));
   }
/*================================================*/
  for (i=0;i<=m-1;i++)
   if ((d[i].l>0 && theta[i].l!=region[i].l) || (d[i].r<0 &&
                                  theta[i].r!=region[i].r)) tag=0;
  return tag;
}

/* See Hansen's book 9.5 method 2 for the function linfun. */
/* It doesn't have to be changed for different problems.   */
plural int linfun(box,ophigh,n,p,m)
plural intvl *box;
double ophigh;
int n,p,m;
```

```
{ plural intvl *middle,tmp,tmpop,u,t,tmpa,tmpd,tmpc;
  int i,j;
  middle=(plural intvl *)p_malloc(m*sizeof(intvl));
  for (i=0;i<=m-1;i++) middle[i].l=middle[i].r=(box[i].l+box[i].r)/2.0;
  tmpop.l=tmpop.r=ophigh;
  for (j=0;j<=m-1;j++)
   { u.l=u.r=0;
     tmp=gsub(getfun(middle,n,p,m),tmpop);
     for (i=0;i<=m-1;i++)
        if (i!=j) u=gadd(u,gmul(gsub(box[i],middle[i]),d[i]));
     u=gadd(u,tmp);
     if (u.l>0 &&  d[j].l==0 && d[j].r==0) return 0;
     else
       { t.l= -DBL_MAX;
         t.r=DBL_MAX;
         tmpa.l=tmpa.r= -u.l;
         if (d[j].l!=0)
           { tmpc.l=tmpc.r=d[j].l;
             tmpc=gdiv(tmpa,tmpc);
           }
         if (d[j].r!=0)
           { tmpd.l=tmpd.r=d[j].r;
             tmpd=gdiv(tmpa,tmpd);
           }
         if (u.l>0 && d[j].l<0 && d[j].r>0)
           { tmpd.l= -DBL_MAX;
             tmpd=gadd(tmpd,middle[j]);
             tmpc.r=DBL_MAX;
             tmpc=gadd(tmpc,middle[j]);
             if (box[j].l<=tmpd.r)
               { if (box[j].r<tmpc.l) box[j].r=box[j].r<tmpd.r?
                                                  box[j].r:tmpd.r;
               }
             else if (box[j].r<tmpc.l) return 0;
                  else box[j].l=box[j].l>tmpc.l?box[j].l:tmpc.l;
           }
         else
           { if (u.l<=0 && d[j].r<0) t.l=tmpd.l;
             else if (u.l>0 && d[j].l<0 && d[j].r<=0) t.l=tmpc.l;
                  else if (u.l<=0 && d[j].l>0) t.r=tmpc.r;
                       else if (u.l>0 && d[j].l>=0 && d[j].r>0) t.r=tmpd.r;
             t=gadd(t,middle[j]);
```

```
            box[j].l=box[j].l>t.l?box[j].l:t.l;
            box[j].r=box[j].r<t.r?box[j].r:t.r;
            if (box[j].l>box[j].r) return 0;
          }
      }
    middle[j].l=middle[j].r=(box[j].l+box[j].r)/2.0;
  }
  return 1;
}

plural int active(box,n,p,m)
plural intvl *box;
int n,p,m;
/* This function reflects the restrictions required by    */
/* each particular problem. It has to be changed for each */
/* problem. tag=0 means certainly infeasible. tag=2 means */
/* certainly feasible. tag=1 otherwise.                   */
{ plural int tag=1;
  tag=2;
  return tag;
}
```